1974

Multiprogrammed Memory Management

Peter J. Denning

G. Scott Graham

Report Number:
74-122

MULTIPROGRAMMED MEMORY MANAGEMENT

Peter J. Denning
G. Scott Graham

Computer Sciences Department
Purdue University
West Lafayette, Indiana 47907
CSD-TR 122
(Rev. Dec. 1974)

Preliminary version of paper to appear in IEEE Proceedings on Interactive Computer Systems, June 1975.

MULTIPROGRAMMED MEMORY MANAGEMENT*

Peter J. Denning,[1] Senior Member, IEEE
and
G. Scott Graham[2]

Abstract: A queueing network is used to show that the page fault rate functions of active programs are the critical factors in system processing efficiency. Properties of page fault functions are set forth in terms of a locality model of program behavior. Memory management policies are grouped into two fixed-partition and three variable-partition classes according to their methods of allocating memory and controlling the multiprogramming load. It is concluded that the so-called working set policies can be expected to yield the lowest paging rates and highest processing efficiency of all the classes.

*Work reported herein was supported in part by NSF Grant GJ-41269.

[1]Address: Dep't Computer Science, Purdue Univ., W. Lafayette, IN 47907.

[2]Address: Dep't Computer Science, Univ. Toronto, Toronto, Ontario, Canada, M5S 1A7.

INTRODUCTION

Eliciting a full, or even adequate, level of performance from a multiprogrammed computer system has proved a difficult goal. Much to the regret of its designers, many a system was put together with but exiguous concern for its ultimate behavior — perhaps because the issues of system organization were more pressing or interesting, or because the complexities of the interactions among demands of different programs for various resources were underestimated. Particularly vexing have been a variety of instability problems, commonly called "thrashing" [W2], and the inability to know which of a myriad of possibilities is the most efficient method of managing a system's memory resources. Two converging streams of research have been increasing our knowledge of analysis and control of system behavior; their eventual confluence will enable the design of new systems whose behavior can confidently be predicted, and may enable improvements in existing systems. The one stream comprises modeling and analysis methods, particularly of networks of interacting queues, that permit studying the effects of competing resource demands both in steady and transient state. Though steady state analysis is more fully developed and provides great insight, full solutions to stability problems await the development of transient state analyses. (See R. Muntz's paper in this issue [M3].) The other stream comprises the study of program behavior and memory management -- that is, the characterization of the relationship between observable patterns of accessing information and demands

on memory and other system resources, and their subsequent use in designing policies of memory management. This paper surveys the present state of knowledge about the interaction of these two streams.

2. SYSTEM ORGANIZATION AND PARAMETERS

A great many contemporary computer systems provide each programmer with a paged virtual address space larger than the main memory space likely to be available when he runs his program. They also provide a file system to permit programmers to store variable numbers of variable objects (files) for indefinite periods of time. We assume that the reader is familiar with the terminology of demand paged virtual memory and of file systems (see, for example, Ref. D3 or S3). Most such systems use multiprogramming, so that main memory will contain a supply of active programs to which the processor can be switched should the one it is working on stop; since a running program typically stops because it requires service from some device other than the processor, multiprogramming improves concurrency in the use of all system resources.

Figure 1 depicts the type of multiprogramming system under consideration here: a network of interacting service stations. The network comprises two main portions: the active network contains the processor and I/O (input/output) stations, while the passive network contains a job queue and policies for admitting new programs to active status. A program is active when in the active network; only when there is it eligible to receive processing and I/O service, and to have pages in main memory. The number of active programs is called the level or degree of multiprogramming; it is denoted hereafter by n or, in the

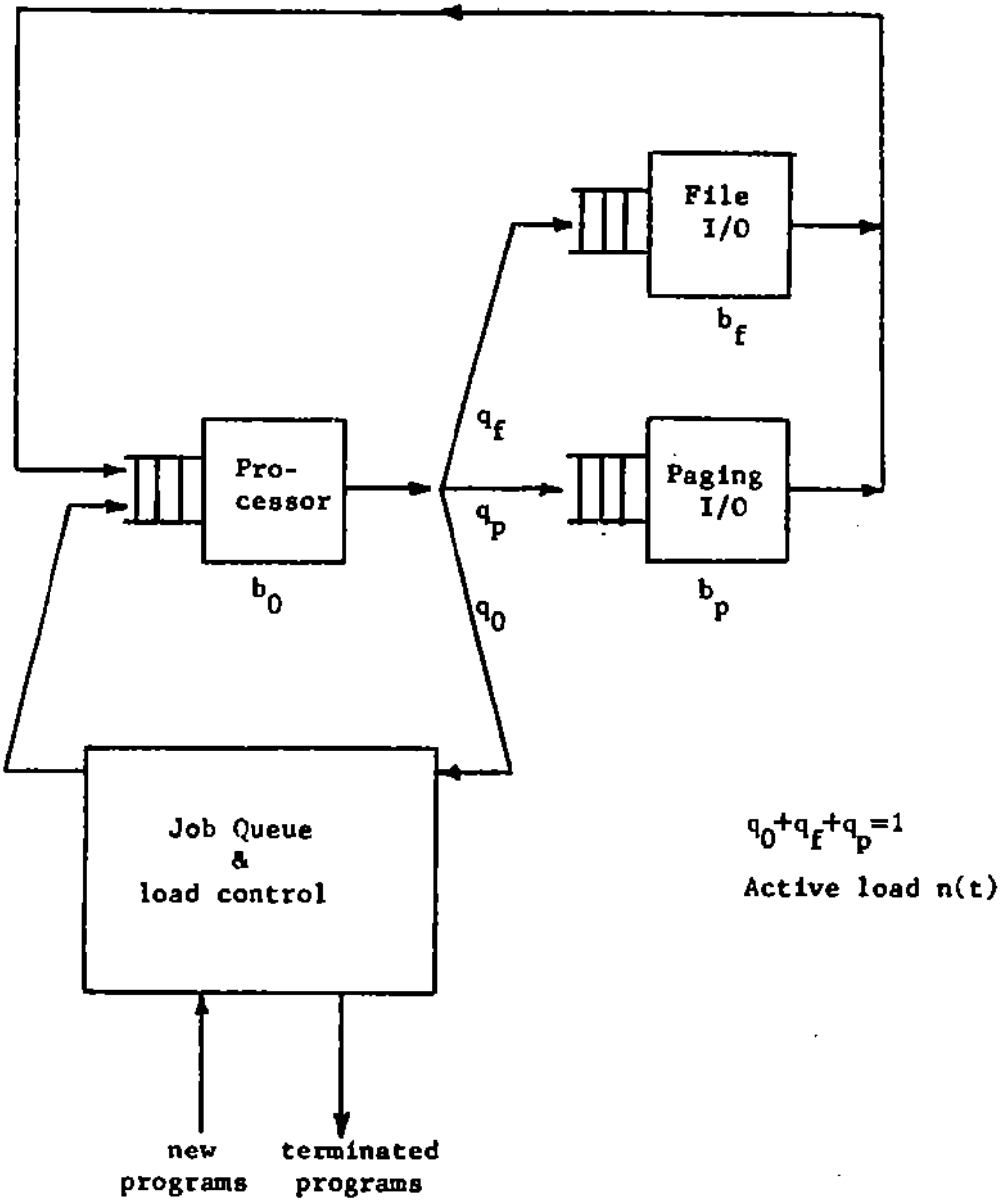

Figure 1. Organization of Multiprogramming System.

In Figure 1, each active program is waiting for service from one of the three stations in the active network: it waits at the file I/O station whenever it requires one or more records of a file to be transferred between a main memory buffer and the file store (usually a disk); it waits at the paging I/O station whenever it requires a page to be transferred between main memory and the paging store (usually a drum); and otherwise it waits at the processor station. The box labelled Job Queue contains a set of enabled programs, a decision policy for activating them, and a "load control" mechanism for controlling n(t). New programs can be submitted from a batch-processing system entry station, a collection of time sharing terminals, or both. (See also B5.)

Inherent in the network of Figure 1 is the notion that an active program alternates between intervals of requiring processor service and intervals of requiring an I/O transaction. Though it is in principle possible for a single program concurrently to be using both the processor and I/O stations, the assumption of no such concurrency is frequently met in practice: demand paging guarantees disjointness of processing and paging I/O, and few programmers ever achieve more than a small percentage overlap between processing and file I/O.

At the completion of a processing interval, a program moves to the file I/O station with probability q_f, to the paging I/O station with probability q_p, or to the inactive state with probability q_0; of course

$$q_f + q_p + q_0 = 1. \tag{2.1}$$

The service rates of the three stations are given by the parameters

b_f, b_p, and b_0; they denote the reciprocals of the mean service times at their respective stations (e.g., $1/b_0$ is the mean length of a processing interval).

The network parameters q_f, q_p, and q_0 are derivable from program parameters. Suppose the total file I/O, paging I/O, and processing requirements of a program are denoted by T_f, T_p, and T_0 respectively. (In our context, T_f and T_0 are entirely intrinsic to a program, whereas T_p is not; how much memory is allocated to a program, or what policy is used to determine which of a program's pages reside in main memory, significantly affect T_p. The system scheduling and memory management policies can cause T_p to vary over an extremely wide range, from considerably smaller than T_0 to considerably larger.) Let a_f denote the rate at which a program requests file I/O; its total number of file I/O requests is therefore $T_0 a_f$, and the total time required to serve all of them is $T_f = T_0 a_f / b_f$. Similarly, if a_p denotes a program's paging rate, the total time it spends on paging is $T_p = T_0 a_p / b_p$. Since on every processor departure, a program chooses independently to leave the active network with probability q_0, the mean number of passes through the processor before leaving is $1/q_0$; and since the mean time per pass is assumed to be $1/b_0$,

$$T_0 = \frac{1}{q_0 b_0} . \tag{2.2}$$

Of the $1/q_0$ passes a program makes on the processor, $(1/q_0)-1$ of them were occasions on which it moved to an I/O station (after the last pass, it exited active status); equating this to the total number of

I/O requests, $T_0(a_f+a_p)$, we find

$$q_0 = \frac{1}{1 + T_0(a_f+a_p)} . \tag{2.3}$$

Together with (2.2), this implies that $b_0 = a_f+a_p+1/T_0$. Since the fraction of processor passes after which a program moves to the file I/O station is q_f, the number of visits it makes there must be $q_f/q_0 = T_0a_f$, which implies

$$q_f = T_0a_fq_0 = \frac{T_0a_f}{1 + T_0(a_f+a_p)} . \tag{2.4}$$

Similarly,

$$q_p = T_0a_pq_0 = \frac{T_0a_p}{1 + T_0(a_f+a_p)} . \tag{2.5}$$

It is clear from (2.3), (2.4), and (2.5) that $q_0+q_f+q_p=1$ as required.

If we now extend T_0, a_f, a_p, b_f, b_p to be averages common to all active programs, we can use the parameter values implied by the above equations to study the average properties of the network.

Now: Define U_0, U_f, and U_p to be the <u>utilizations</u> (fraction of time busy) of the three stations, for given load and parameter settings. In equilibrium, the mean flow of programs out of the file I/O station must be U_fb_f programs per unit time; out of the paging I/O station, U_pb_p; and out of the processor, U_0b_0. Moreover, a fraction q_f of U_0b_0 must be input to the file I/O station and, in equilibrium, the input flow must be the same as the output flow there; hence

$$U_fb_f = U_0b_0q_f = U_0b_0(T_0a_fq_0) = U_0a_f \tag{2.6}$$

where eqs. (2.4) and (2.2) have been used to simplify. Define the

relative utilization of the file I/O station,

$$R_f = \frac{U_f}{U_0} = \frac{a_f}{b_f} . \tag{2.7}$$

Similarly, for the paging I/O station,

$$R_p = \frac{U_p}{U_0} = \frac{a_p}{b_p} . \tag{2.8}$$

The relative utilization of the processor station is of course $R_0=1$. It is important to note that R_p can be interpreted as the ratio of the mean paging I/O service time to mean uninterrupted processing interval between page faults. (An analogous statement can be made for R_f.) In other words, if $S = 1/b_p$ is the mean paging I/O service time, and $L = 1/a_p$ is the mean length of execution interval between page faults (assuming the main memory access time is used for the unit of virtual time), then (2.8) can be rewritten

$$R_p = \frac{S}{L} . \tag{2.9}$$

The foregoing discussion assumes that all active programs have the same system parameters. If they do not, we can use as an approximation suitable averages over all active programs. For example, if a sequence of k successive page faults (from programs of different characteristics) terminate interfault intervals of expected lengths $L_1,\ldots,L_k$, we can use $L = (L_1+\ldots+L_k)/k$ in (2.9). In reality, the processor utilization is a function of all the intervals $L_1,\ldots,L_k$, not just their average. However (as we have verified by simulations), the use of the average L appears to give predictions of utilization within a few per cent of the true utilization, and thus we felt justified in using the simpler

analysis based on averages over the set of active programs. Nonetheless, the reader should keep in mind that the use of these averages in fact constitutes an approximation.

Concerning utilizations, a few points should be noted. First, the ratio R_f depends only on the intrinsic program parameter a_f and the (fixed) file I/O station rate b_f; it cannot be affected by memory management policies. In contrast, the ratio R_p depends on the paging rate a_p, which can be controlled by the system. Therefore, in our context, the paging rate is the critical parameter. Second, the relative utilizations R_f, R_p, and R_0 do not depend on the load (level of multiprogramming). However, the absolute utilizations do. Under general assumptions, one can show that

$$U_0 = U_0(n, R_f, R_p), \tag{2.10}$$

that is, the absolute processor utilization depends only on the load and the relative utilizations [C4, B5, B6]. Once U_0 is found, the other utilizations can be obtained from $U_f = U_0 R_f$ and $U_p = U_0 R_p$. Third, if R_f and R_p are fixed, U_0 must be an increasing function of load: for a new active program must increase the absolute utilization of any station at which it queues, and, because the utilizations are in fixed ratios, all other absolute utilizations must increase. Therefore,

$$U_0(n+1, R_f, R_p) > U_0(n, R_f, R_p). \tag{2.11}$$

Fourth, as load increases, the utilization of the station having the maximum relative utilization must approach 1 at least as fast as the others. Let

(2.12) $$R_m = \max[R_f, R_p, R_0];$$

since $U_0 = U_m/R_m$, the fact of U_m approaching 1 fastest implies

(2.13) $$U_0(n, R_f, R_p) \leq \frac{1}{R_m},$$

with near equality for large enough n. Since $R_m \geq 1$, the maximum possible value of U_0 may in fact be less than 1. This shows that the designer of a systemwhich apparently is unable to achieve processor utilization 1 cannot immediately conclude that an improvement in the memory management policy will increase U_0. A slow file I/O station, or excessive rate of file I/O requests, can cause $R_m = R_f > 1$: the file I/O station, rather than the paging I/O station, in this case limits processor utilization. Usually, however, adequate buffering keeps $R_f \leq 1$, so that reductions in R_p are likely to improve performance.

The properties above show what happens when load is changed and other parameters are held fixed. In studying memory policies, it is frequently possible to vary the paging rate while holding load and other parameters fixed. From the above, it follows that

(2.14) $$U_0(n,R_f,R'_p) \geq U_0(n,R_f,R_p), \quad R'_p \leq R_p.$$

In words, changing the paging rate from a_p to $a'_p \leq a_p$ cannot decrease processor utilization. For if reducing R_p were to cause U_0 to decrease, then $U_f = U_0 R_f$ would decrease as well — implying a decrease in the utilizations of <u>all</u> stations, which is patently impossible without reducing the load.

Since the throughput rate of the active network of the system is the flow out, viz.,

$$\lambda = U_0 b_0 q_0 = \frac{U_0}{T_0} , \tag{2.15}$$

it follows that increasing processor utilization for a given level of multiprogramming improves the system's ability to complete work at that load level. For multiprogramming level n, Little's formula tells that the response time in the active set is

$$W = \frac{n}{\lambda} = \frac{nT_0}{U_0} , \tag{2.16}$$

that is, increasing U_0 without changing n will decrease response time. Therefore, decreasing the relative utilization of the paging I/O station by an improvement in memory policy without changing the load will concommitantly increase throughput and decrease response time. For this reason, processor utilization is a suitable measure of performance.

The previous observations about processor utilization do not consider what happens when an increase in load implies an increase in page fault rate, on account of programs having less space available. Systems under memory constraint exhibit an optimum level of multiprogramming, n_0, U_0 being maximum at n_0 [B5]. The reason is that overall paging rate a_p is an implicit function of load, with $a_p(n+1) \geq a_p(n)$. However, for $n<n_0$, the increase in paging is unable to offset the increase of utilization effected by increased load; but for $n \geq n_0$, paging increases more rapidly and utilization decreases. An extreme case will illustrate. Suppose total main memory is M pages and each active program receives space $x = M/n$ under load n. Take $R_f = 1$ and R_p to be the step function

$$R_p = a_p(x)/b_p = \begin{cases} 100, & x \le x_0 \\ 1/100, & x > x_0 \end{cases}$$

In other words, these programs page at a high rate when their memory allocations are small and at a low rate otherwise. This implies that the processor utilization has the form (cf. eq. (2.13)):

$$U_0(n,R_f,R_p) = \begin{cases} U_0(n,\ 1,\ 1/100) \le 1/R_m = 1, & n < M/x_0 \\ U_0(n,\ 1,\ 100\ ) \le 1/R_m = 1/100, & n \ge M/x_0 \end{cases}$$

This is suggested in Figure 2. The optimal degree of multiprogramming is $n_0=M/x_0$. The effect suggested here, known as thrashing, is not usually so abrupt as this example shows. However, in many practical situations changing the load from n_0 to n_0+1 or n_0+2 is sufficient to cause a serious drop in utilization.

The optimal level of multiprogramming can vary from one set of active programs to another, because page fault rates vary among programs: thus $n_0=n_0(t)$. To avoid thrashing, it is necessary to include a load control mechanism in the system scheduler (Job Queue in Fig. 1), whose purpose is to adjust dynamically the level of multiprogramming so that most of the time $n(t) \le kn_0(t)$ for some small constant $k \ge 1$. Even a simple limit N on n(t) may not successfully control thrashing, unless N has been set low enough so that the event $kn_0(t) < N$ is unlikely -- but then the system is probably operating at a significantly suboptimal load a goodly portion of the time. Load controls which attempt to maintain $n(t)=n_0(t)$ and which thereby keep the system operating at top efficiency will be discussed later. (See also R2 and W2.)

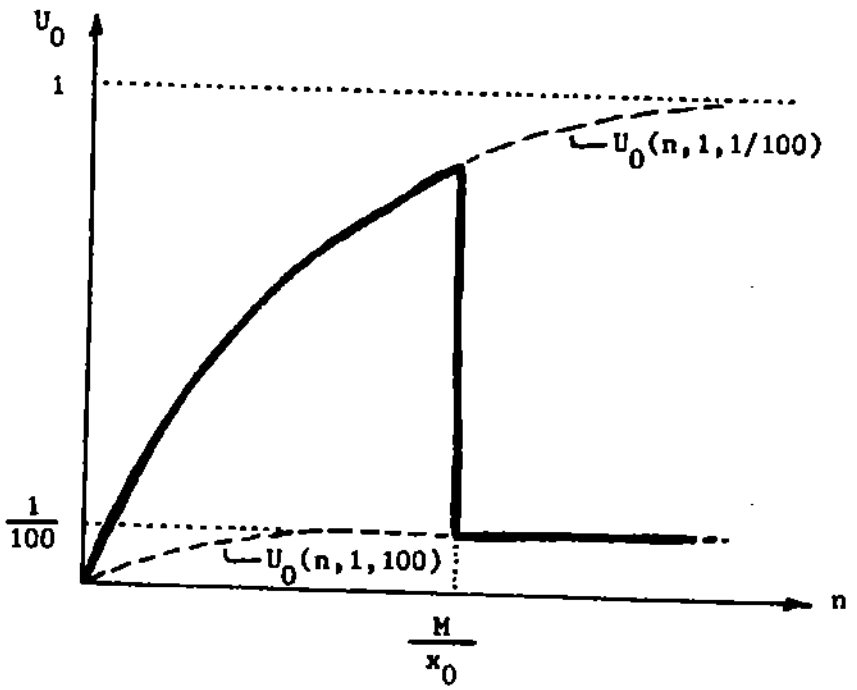

Figure 2. Thrashing Effect.

To summarize: We have examined a network representation of the resources used by active programs in a typical multiprogramming environment. The purpose was to establish that the page fault rate is the critical parameter, and that memory policy changes that improve it without changing load or other system parameters can be expected to improve processor utilization, increase throughput, and decrease response time. To show whether a proposed change in the memory policy will improve processing efficiency, it is usually sufficient to show that the change does not increase any program's paging rate, or equivalently that it decreases the relative utilization of the paging I/O station. We showed also that there is an optimum load, that a load control mechanism is required to prevent thrashing, and that load control must be coupled to the memory policy.

3. PROGRAM BEHAVIOR AND PARAMETERS

A program in execution will generate a sequence of references (known as an address trace) to information in its virtual address space. The reference string of the program is a sequence

$$\underset{\sim}{R} = r(1)\, r(2) \ldots r(k) \ldots r(K), \tag{3.1}$$

in which r(k) is the number of the page containing the virtual address referenced at time k, where $k = 1,2,\ldots,K$ measures execution time, or virtual time. The pages the program has present in main memory constitute its resident set; the resident set just after the k^{th} reference is denoted

by Z(k), and its size (in pages) by z(k). A page fault occurs at virtual time k if r(k) is not in Z(k). Under the assumption of demand paging, Z(k+1) is the same as Z(k) plus r(k) less any pages of Z(k) replaced (i.e., removed from main memory) by the memory policy; moreover, $z(k+1) \leq z(k)+1$. The memory policy thus determines the sequence of resident sets Z(1)Z(2)...Z(K) that arises while processing a reference string $\underset{\sim}{R}$ and, hence, the paging rate experienced by the program generating $\underset{\sim}{R}$.

Let $t_1, t_2, \ldots, t_K$ denote the (real) time instants at which the references of a reference string $\underset{\sim}{R}$ commence. The resident set at time t, where $t_{k-1} \leq t < t_k$, is the same as that at time t_{k-1}, less any pages which have been replaced; thus

$$Z(t_k) \subseteq Z(t) \subseteq Z(t_{k-1})+r(k-1) \quad . \tag{3.2}$$

It is important to keep clear the distinction: the behavior of a given program is formulated with respect to its virtual time, whereas the behavior of a system is formulated with respect to real time.

For reasons already discussed, the page fault rate function is important in any study of memory management. Denoted by f(A,x), this function gives the expected number of page faults generated per unit of virtual time when a given reference string $\underset{\sim}{R}$ is processed by memory policy A, subject to main memory space constraint x. Since most of the results depend only on properties which, being common to most fault-rate functions, are relatively independent of the particular $\underset{\sim}{R}$ that arises, $\underset{\sim}{R}$ will not be shown as an explicit parameter of these functions; however, the dependence should not be forgotten altogether.

For the case of fixed memory allocation, the space constraint x is interpreted to mean that the resident set sizes must satisfy $z(k) \leq x$ for all virtual times k. For the case of variable space allocation, the space constraint x is interpreted to mean that the average resident set size is x:

$$x = \frac{1}{K} \sum_{k=1}^{K} z(k); \tag{3.3}$$

it is assumed that the policy A has parameters which can be adjusted so that (3.3) can be satisfied for a range of choices of x. Examples of both fixed and variable space policies will be considered below.

Examples of commonly studied fixed-space policies include: LRU (least recently used) which, at a page fault time, replaces the least recently referenced page of the resident set; FIFO (first in first out) which, at a page fault time, replaces the longest resident page; RAND (random) which, at a page fault time, replaces a randomly chosen page from the resident set; and OPT (optimal) which, at a page fault time, replaces the resident set page that will not be referenced again for the longest time. Of these, OPT cannot be implemented (it requires foreknowledge), FIFO is simplest to implement (it requires arranging the resident set pages in an order-of-arrival queue), and LRU is the most robust, providing consistently the lowest (of nonOPT policies) fault rate over the widest class of reference strings [B1]. Although OPT is not implementable, it can be used a posteriori to compare various algorithms against optimum; and its principle — choosing for replacement the page with maximum "forward distance" — can easily be used to construct

reference strings for which LRU is optimal or approximately so (see Appendix 1). If the memory policy A is a member of the large class of "stack algorithms" [C4, M1], the fault rate function f(A,x) is non-increasing in x for every reference string and may be computed by a highly efficient procedure. Of the above, all but FIFO are stack algorithms.

Much of our attention will be directed toward the LRU policy, or procedures resembling it. Associated with an instance of this policy is a dynamic list known as the LRU stack, that arranges the referenced pages from top to bottom by decreasing recency of reference. At a page replacement time, the LRU policy chooses the lowest ranked page in the stack; therefore, the contents of an x-page resident set must always be the pages occupying the first x stack positions. When a page is referenced, the stack is updated by moving the referenced page to the top and pushing down the intervening pages by one place. The position at which the referenced pages was found in the stack before being promoted to the top is called its stack distance. A page fault occurs in an x-page resident set at a given reference, if and only if the stack distance of that reference exceeds x. These ideas form the basis of an efficient procedure for computing the fault rate function f(LRU,x) by counting stack distances in a reference string (see Appendix 1). Figure 3 shows a typical such function. It has the terminal values f(LRU,0) = 1 and f(LRU,N) = N/K for an N-page program and reference string of length K. For large K, the function is typically convex, which is considered a manifestation of program locality (see below).

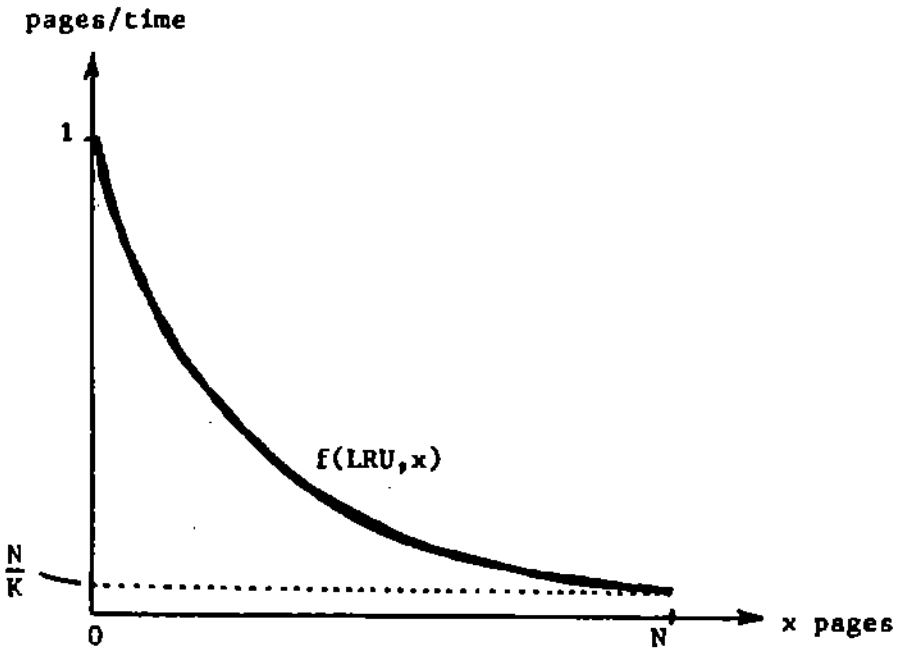

Figure 3. Typical LRU fault rate function.

Though powerful, analysis of given reference strings under fixed space policies does not account for the mechanisms by which programs generate reference strings; moreover, the procedures do not readily extend to the analysis of variable space policies. To deal with this, a model is useful. Regard a program's execution time as being partitioned into a sequence of phases, a phase being an interval of constant memory requirement. Similarly, the program's address space is partitioned into segments, a segment being a named block of contiguous addresses. A given segment is considered "active" in a given phase if processing of that phase requires the presence of that segment in main memory -- i.e., in the resident set. The set of all segments active in a given phase is called the locality set, or locality, of that phase; the locality set at a given instant of real time is the same as that of the phase in progress at that time [S5,S7]. The validity of this abstraction has been verified over and over again, for example in the experiments of Rodriguez-Rosell [R1], of Hatfield and Gerald [H1], or of Ferrari [F1]: it is always observed that many distinct and readily-identified phases exist, that during each an often-small subset of the program's segments is active, and that the locality sets are often disjoint and of highly variable sizes.

Though not of direct concern here, the distribution of segments among pages can have a significant effect on locality [F1,H1]. In case a large segment is allocated among several smaller pages, its activity implies that of all its pages, so that the original locality properties remain observable. In case a number of small segments are allocated on

one larger page, the assignment can be critical: scattering the segments of one locality among many pages will effectively mask the locality properties of the original program, making it appear as if the locality set of every phase -- measured now in pages -- is very nearly all the address space.

The notion of localities and program phases is somewhat more viable in the context of generating page reference string [S5,S7] than in the context of designing memory policies, simply because prior knowledge of localities and phase boundaries is not available in the latter context. Instead, a memory policy must include some method of measuring or estimating the locality of a program at each instant, and the estimator thus defined can be used to specify the content (and the size, if that is adjustable) of the resident set of a program. The generic term working set is usually used to denote an estimator of a locality set. Just as there is a wide range of fixed space paging algorithms, there is a wide range of locality estimating techniques and a range of variable space policies based on them. A characterization of this range will be presented in the next Section.

Perhaps the most well known locality estimation method is the moving-window working set (WS). Its analysis methods are even more fully developed than those for fixed space paging algorithms [C4,D1, D2,D4,G2,P1,S4]. For a parameter T known as the window size, the working set W(k,T) of a program at virtual time k is the collection of pages referenced by that program in the T references preceding and including the one at time k (i.e., $W(k,T) = \{r(k-T+1),\ldots,r(k)\}$); if $k<T$, $W(k,T) = W(k,k)$. The size of the working set is denoted by w(k,T).

If $\{t_k\}$ are the real time instants corresponding to virtual times $\{k\}$, and $t_k \leq t < t_{k+1}$, define $W(t,T)=W(k,T)$. The missing page rate $m(T)$ is the rate at which new pages are entering the working set. Under the pure working set memory policy (WS), which allocates the resident set as $Z(t)=W(t,T)$, $m(T)$ becomes the page fault rate. The mean working set size is denoted $s(T)$; it is an increasing, concave function whose slope may be interpreted as $m(T)$ (see Appendix 1). A fault rate function $f(WS,x)$ giving directly the relation between (mean) space and paging rate can be defined parametrically by setting

$$f(WS, s(T)) = m(T), \quad T=0,1,2,\ldots \tag{3.4}$$

All the functions $m(T)$, $s(T)$, and $f(WS,x)$ can be computed efficiently (see Appendix 1).

Figure 4 shows the mean working set size function $s(T)$, the missing page rate function $m(T)$, and the construction of the fault rate function $f(WS,x)$. The curve $s(T)$ approaches a value s_{max}, which it attains for some $T \leq K$ where K is the reference string length. In general $s_{max} \leq N$, N being the program size; however, s_{max} need not equal N since the program need not reference some of its pages until later phases, whereupon early working set sizes must be less than N. The function $m(T)$ is decreasing to the value N/K. The fault rate function $f(WS,x)$ is defined only for $0 \leq x \leq s_{max}$, with terminal values $f(WS,0)=1$ and $f(WS,s_{max}) = N/K$; it is decreasing since $s(T)$ is increasing and $m(T)$ decreasing.

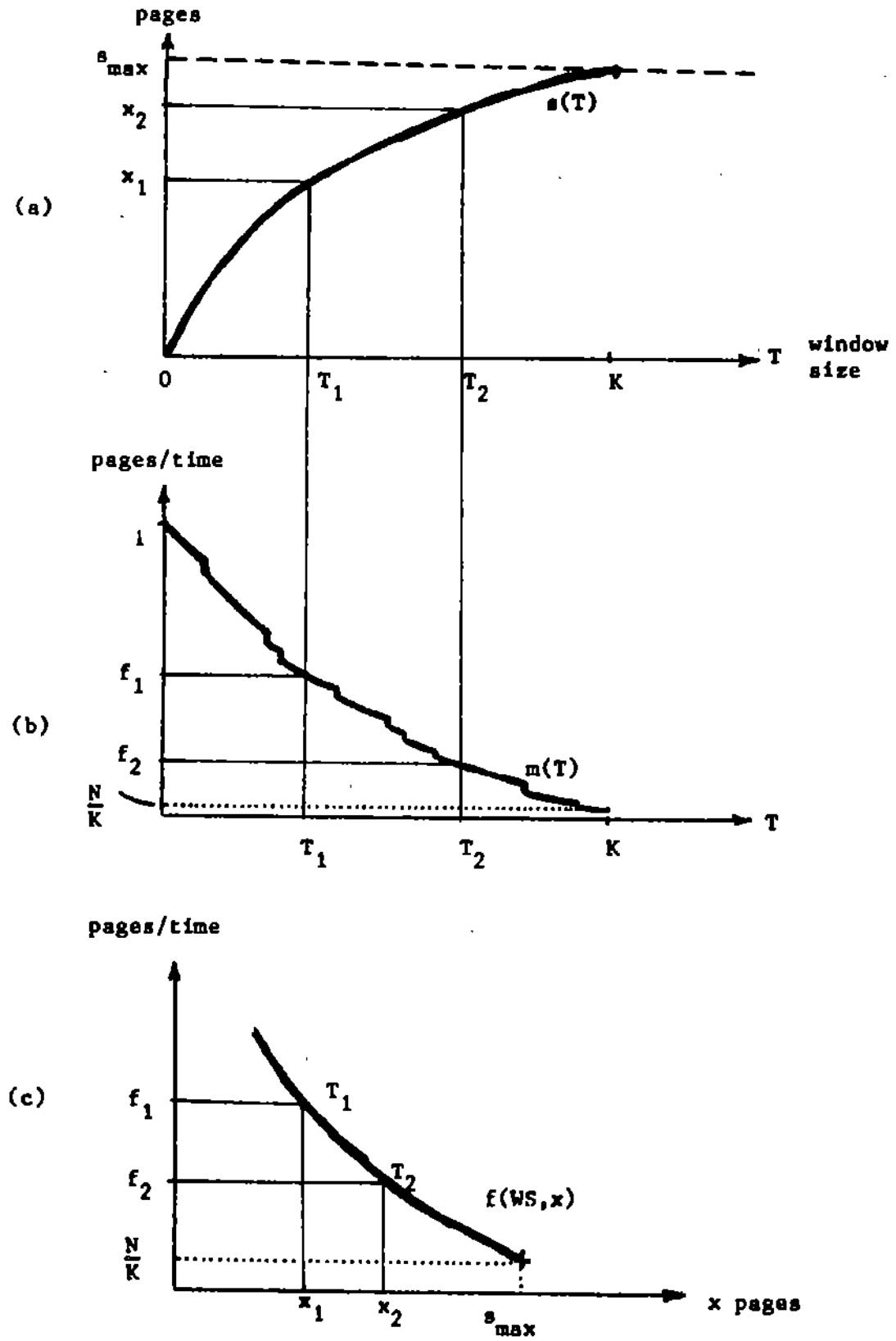

Figure 4. Working set properties, and construction of fault rate function.

Figure 5 shows a typical comparison of f(WS,x) and f(LRU,x). Define the point x_0 as the smallest space for which $x \geq x_0$ implies $f(WS,x) \leq f(LRU,x)$ -- that is, WS is at least as good as LRU. The point x_0 appears not to exceed the mean locality set size of the reference string. Thus a program with one phase and one locality set will have $x_0 \simeq s_{max} \simeq N$, while one with many phases and a wide variance among locality set sizes will tend to have x_0 much smaller than s_{max} and N. The reason is that WS is able to adapt its resident set to be the current locality set estimate, having little or no paging whenever the window is contained wholly in a phase, whereas LRU will produce streams of page faults in those phases whose locality exceeds the size of its resident set. This behavior will be observed even for reference strings over which LRU is optimal, and for reference strings processed by the optimal fixed space policy OPT: it is a direct result of the variability of locality size.

Experiments by Prieve and Fabry [P1,P2] indicate that differences of 30% or more between the LRU and WS curves in the range $x > x_0$ occur frequently, showing that important variations in locality size are significant in program behavior. (See Appendix 1 for examples.) Further experiments demonstrate that an optimal variable space algorithm could in principle produce another 30% or more improvement over WS (60% or more over LRU), showing that the working set is not a perfect estimator of locality [P3]. (However, like the fixed-space optimal algorithm, OPT, the one studied by Prieve requires foreknowledge of the reference string. Its primary interest is in assessing how effective a locality estimation procedure is.)

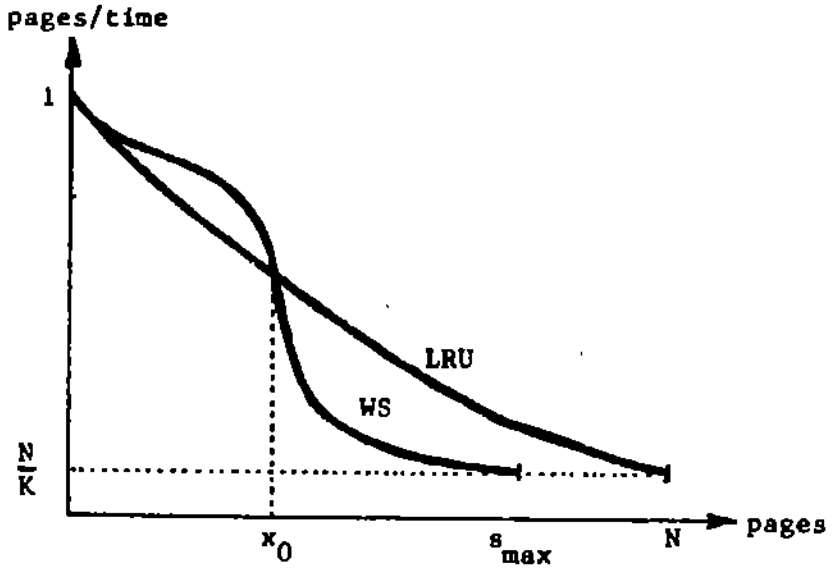

Figure 5. Comparison of LRU and WS.

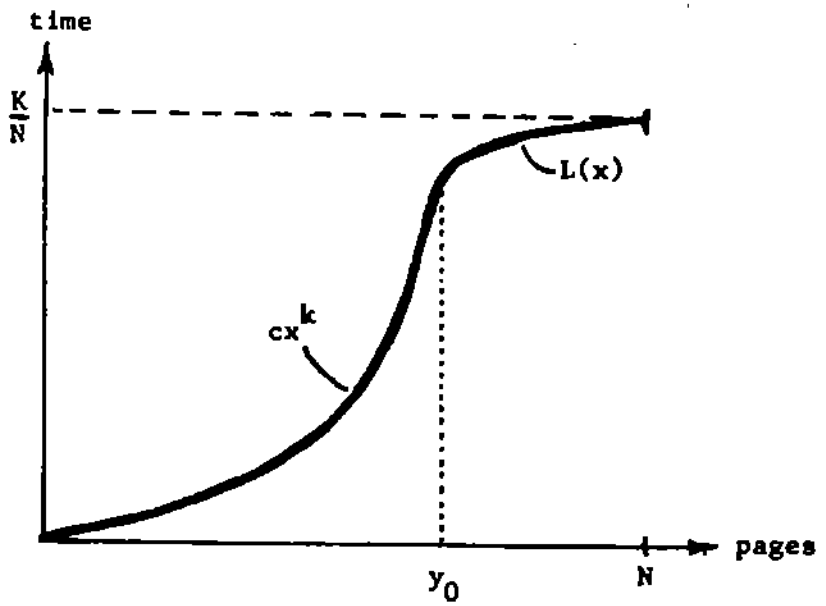

Figure 6. Lifetime function.

Another measure of page faulting is the lifetime function L(x), which gives the mean virtual time between page faults when a reference string is processed under a given memory policy with space constraint x. (See C1, B2, B3.) It is defined simply as

$$L(x) = \frac{1}{f(x)}, \tag{3.5}$$

where f(x) is a fault rate function. Figure 6 shows a typical lifetime function for the LRU policy. There is usually a value y_0 such that the lifetime function is convex for $x \leq y_0$ and concave for $x > y_0$. For some fixed space policies, the convex part can be approximated by cx^k with $1.5 < k < 2.5$ [B2,B3,C1,S6]; no one has ventured approximations yet for the convex part of L(x) for variable space policies. The convex/concave shape is characteristic of many (but not all) lifetime functions. For fixed space policies, y_0 is approximately the size of the largest locality set, whereas for the WS policy it tends to be approximately the average (over virtual time) of the locality set sizes. (See Appendix 1.) The properties of the lifetime function have been used to demonstrate efficiency increases in certain cases, where such increases could not be deduced directly from the properties of the fault rate function [C1,G2,S6].

To summarize: A program's page reference string is an observable quantity, from which one can compute fault rate functions for various memory policies, notably LRU and WS. The abstractions of phases and localities can be used to explain the relative behaviors of programs under LRU and WS policies. The lifetime function, which is the reciprocal of the fault rate function, characteristically exhibits a convex/concave shape.

The existence of a convex region in the lifetime function will be used in the next section to deduce that certain variable space (nonWS) policies can produce a net reduction in paging rate; together with the results of Section 2, this implies a net improvement in system performance.

4. CLASSIFICATION OF MEMORY POLICIES

Denote by $P_1,\ldots,P_n$ the set of active programs during a time interval in which the level of multiprogramming is fixed ($n=n(t)$). Associated with P_i at time t is its resident set $Z_i(t)$, containing $z_i(t) \geq 1$ pages. The configuration of memory is represented by a <u>partition vector</u>

$$\underset{\sim}{Z}(t) = (Z_1(t),\ldots,Z_n(t)). \tag{4.1}$$

The <u>partition size vector</u> is

$$\underset{\sim}{z}(t) = (z_1(t),\ldots,z_n(t)), \tag{4.2}$$

in which

$$z_1(t)+\ldots+z_n(t) \leq M \tag{4.3}$$

at every time instant t, where M is the size of the main memory. The <u>reserve memory</u> is that portion unused by any active program; its size at time t is

$$R(t) = M - \sum_{i=1}^{n} z_i(t). \tag{4.4}$$

As has been noted, a memory policy can be regarded as including a method of estimating program locality sets. The estimates thus determined are used to specify the content (and size, if adjustable) of each program's resident set. Figure 7 suggests a classification of memory policies based on the method used to estimate the locality. It will be used as the basis for the ensuing discussion of memory policies. Our objective is showing why performance improvements can be expected under a policy improvement corresponding to a rightward change along the bottom of the diagram in Figure 7.

When main memory allocation is controlled by the programmer, who inserts into the program commands that move information in and out of main memory, memory management is said to be manual. The viability of this type of management is usually limited to systems in which the resident set size is fixed and known in advance by the programmer, who is then in a position to optimize information placement and flow with respect to that resident set size. In contrast, memory management handled by the system is said to be automatic.

Manual memory management has fallen out of favor for a variety of reasons. One is simply mounting experience that properly designed automatic management mechanisms (e.g., virtual memories) can perform at least as well for large programs as carefully planned overlays [S2]. Another is the use of multiprogramming and multiplexed resource allocation, which rob the programmer of the key assumption that a resident set of known size will be continuously available to him. In a multiprogrammed environment, each active program P_i cannot be guaranteed good performance even if it has complete control over its resident set $Z_i(t)$, because

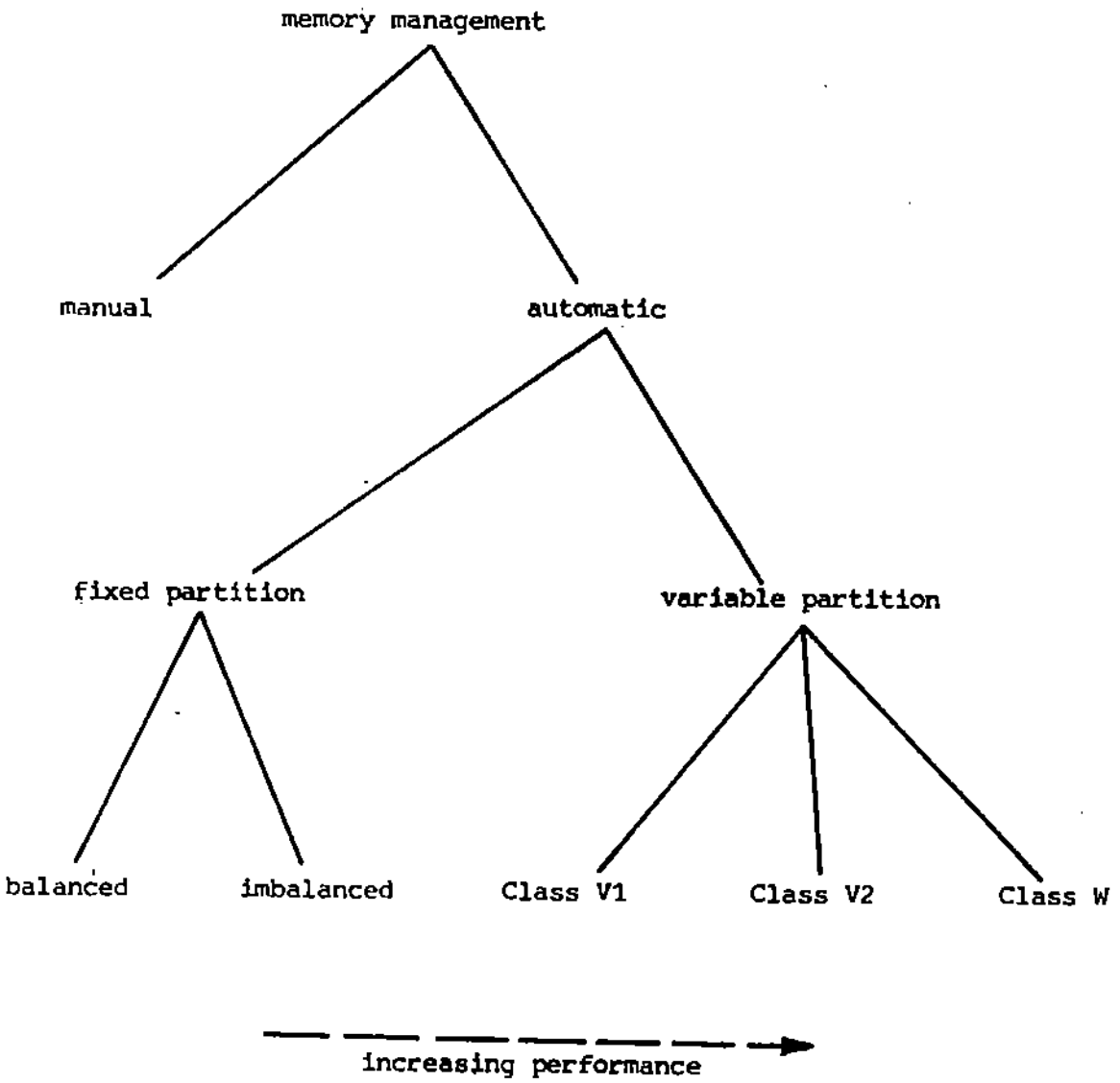

Figure 7. Classification of memory policies.

attempted local optimizations need not imply that the entire system is optimized. The problem is that the individual who programmed P_i does not have access to information about P_j ($j \neq i$) and is therefore not in a position to optimize his performance in relation to the system's; moreover, there is no guarantee he would use this information properly even if he did have it. Therefore, multiprogrammed memory management is always automatic.

Policies of automatic memory management can be grouped in two categories: fixed partitioning and variable partitioning. The latter has already been defined, in terms of a time varying partition vector $\underset{\sim}{Z}(t)$; techniques of varying the partition will be discussed below. If the resident set size $z_i(t)$ is a fixed constant z_i for all t during which P_i is active, then the size vector $\underset{\sim}{z}(t)$ is constant during any interval in which the set of active programs is fixed; this is known as the fixed partition approach. In case the entire address space A_i of P_i can fit in the allocated space of z_i pages, the resident set is also fixed: $Z_i(t) = A_i$. Otherwise, if z_i is smaller than the size of A_i, a replacement policy must be used to define what subset of A_i constitutes $Z_i(t)$; note in this case that $Z_i(t)$ varies even though z_i does not. In case $z_i = M/n$ for each i, $\underset{\sim}{Z}(t)$ is called a balanced partition, or equipartition.

Imbalanced partitions are capable of better processing efficiency, if only because they permit the flexibility of allocating programs with larger locality sets more memory. However, even if all programs have identical locality properties, it frequently happens that any imbalanced partition is more efficient than a balanced partition (see Appendix 2 and C1, G2, S6).

In either the fixed or variable partition approach, demand paging is ordinarily used to acquire a program's pages into main memory while that program is active. In case latency time at the paging I/O station is a problem, some form of swapping may be used to load a resident set at the beginnings and ends of a program's active intervals.

Arguments in support of fixed partitioning are of two types. One is founded on a belief that memory availability in a system, and the memory requirements of any given program, can be predicted prior to program processing. The other is the apparent low overhead of implementation, since partition changes occur as infrequently as possible -- viz., when the set of active programs changes. The first argument is weak for the same reasons that arguments for manual overlays are. The second argument's weakness is revealed when one accounts for changing locality in a program. Consider for a moment the behavior of a fixed partition $\underset{\sim}{z}$ when the set of active programs $P_1,\ldots,P_n$ each has a large variance in locality set size across time. Because the partition is fixed, there is no way to reallocate pages from z_i to z_j at a time when P_i's locality is smaller than z_i and P_j's locality is larger than z_j, when clearly such a reallocation would not degrade P_i's performance but would improve P_j's. Coffman and Ryan have analyzed this effect, and have concluded that the variance in locality size from one program phase to another is ordinarily large enough to produce a gain in memory utilization so significant that it recovers the cost of implementing a variable partition several times over [C3]. Put another way, there is a hidden overhead in the fixed partition -- severe loss of storage utilization for programs with wide variance of locality size -- which when accounted for significantly diminishes the attraction of fixed partition strategies.

Within the class of variable partition strategies, one may identify at least three subclasses, according to whether there is no correlation, weak correlation, or high correlation with locality changes of programs:

Class V1 - The partition $\underset{\sim}{Z}(t)$ is varied, but with no explicit correlation to the reference patterns of the active programs.

Class V2 - Variation in $\underset{\sim}{Z}(t)$ is explicitly correlated with the activities with which active programs reference pages in their resident sets, but there is no explict attempt to identify locality sets and protect them from preemption.

Class W - The resident sets $\underset{\sim}{Z}(t)$ are maintained in one-to-one correspondence with working sets (estimates of the locality sets) of active programs.

It should be noted that Class W policies are intended to be precisely the "working set policies" [D2,D3].

As noted in Section 2, the efficiency of a multiprogrammed computer system depends on a load control mechanism keeping the system away from thrashing. The objective is to control the level of multiprogramming n(t) by activating or deactivating programs so that most of the time

$$n(t) \leq k n_0(t) \tag{4.5}$$

where $n_0(t)$ is the optimum level of multiprogramming and $k \geq 1$ is a small constant. Class W policies have an inherent load control: they will force the deactivation of a program at a page fault time when the memory reserve R(t)=0; and they will defer the activation of a new program whose initial working set is estimated at size z, until

(4.6) $$z \leq R(t)-H, \quad H > 0.$$

The parameter H is adjusted so that the rate of program deactivations caused by R(t)=0 at a page fault time is low [C3,R3]. To the extent that a W-policy is successful in estimating localities, it will tend to have n(t) approximately $n_0(t)$ (see Appendix 3). In contrast, V1 and V2 policies, which have by definition no direct way of estimating locality, must necessarily use some cruder form of load control. A Typical control for these cases is a preestablished limit N on the allowable level of multiprogramming, sometimes with an adjustment of the limit N inversely with the system paging rate. To keep the thrashing probability low, it is necessary to set N so that the event $kn_0(t) < N$ is unlikely — which implies that the system spends most of its time operating at a suboptimal level of multiprogramming. Thus, even if a V2-policy is successful in keeping locality sets resident, it will tend to be less efficient than a W-policy. Finally, one should expect V1-policies to be less efficient even than V2-policies, since they have no mechanism at all for tending to reallocate pages from resident sets that are larger than their contained locality sets to resident sets that are smaller: because they thus make poorer overall use of storage, they cannot maintain as high a level of multiprogramming at a given level of paging as a good V2-policy and (by eq. (2.11)) their processing efficiency will be lower. The empirical evidence supporting this ranking of the classes is discussed next.

Class V1 Policies

Though it may not be obvious that varying a partition without correlation to program behavior can increase system processing efficiency, V1-policies are capable of improving over fixed partition policies. This was first observed in a study of the so-called biasing discipline by Belady and Kuehner on the M44 system [B2,B3]. According to this discipline, a "favored state" of execution is passed cyclically among the active programs. A given program remains in the favored state until the system has experienced p page faults (p is a parameter). While favored, a program is granted new pages on demand for its resident set and is exempted from replacements; thus its resident size can increase by as many as p pages during its favored interval. A system throughput increase in the range 10-15 per cent over an (approximate) equipartition using FIFO replacement is reported. Belady and Kuehner suggest (but do not prove) that the performance improvement derives from the convexity of the lifetime function. In Appendix 2 we show that, given a fixed partition $\underset{\sim}{X} = (X_1,\ldots,X_n)$, there exists a V1-policy under which the fault rate of each active program P_i satisfies

$$f_i(X_i) > \bar{f}_i > f_i(\bar{x}_i), \tag{4.7}$$

where $\bar{f}_i$ is the mean virtual time fault rate of P_i, and $\bar{x}_i$ is the mean resident size in the virtual time of P_i. The lefthand inequality assumes that the lifetime function is convex over the range of memory allocations used; the righthand inequality assumes that the fault rate function is convex. One can show $\bar{x}_i > X_i$ even if the righthand inequality above is false, directly from the assumption that f_i is a decreasing function.

Thus the fixed partition $(\bar{x}_1,\ldots,\bar{x}_n)$, which would be yet more efficient that the V1-policy, is hypothetical — it cannot be implemented, since $\bar{x}_i > X_i$ implies $\bar{x}_1+\ldots+\bar{x}_n > M$.

Analyses by Spirn [S6], Ghanem [G2], and Chamberlin et al. [C1] have given further information about partition policies. These authors worked with lifetime functions of the type discussed earlier (see Fig. 6). They discovered that processing efficiency may be increased relative to an equipartition, by amounts comparable to those observed by Belady and Kuehner, simply by using an imbalanced partition. No variable partition is needed. (See Appendix 2.) Spirn showed further that the equipartition may be worst possible. Moreover, Ghanem showed that this result may depend on the lifetime function's being "sufficiently convex" for $x<y_0$ (cf. Fig. 6). That is, for lifetime functions of the form $L(x) = cx^k$ $(x \leq y_0)$, it was necessary that $k>a$, where a is a constant depending on program and system parameters; typically $1 < a < k < 2.5$. Ghanem found a stronger result: when the lifetime is insufficiently convex, the equipartition is optimal.

It thus appears that two factors may have combined to produce the effect observed by Belady and Kuehner. One is that their policy always kept the system in some imbalanced partition. By changing the partition only at page fault times they not only kept the overhead of their policy to a minimum, but they distributed the improvements uniformly among the active programs. The other factor is the space variation acting in the virtual time of the programs producing the relation (4.7).

All these analyses and observations lead inescapably to the conclusion that lifetime functions of programs are significantly non-linear, a fact which has yet to be reconciled by a linear assumption to which one may be led on superficial inspection of a recent paper [S1]. (See also D6.)

Class V2 Policies

An approach to memory management commonly used in operating systems extends the idea of a fixed space replacement policy to multiprogramming simply by applying the replacement rule to the entire contents of memory, without identifying which program is using any given page. For example, all resident set pages (of $Z_i(t)$ for each i) can be placed on a single ("global") LRU stack; whenever an active program runs, it will presumably reference its locality set pages and move them to the top of this LRU stack. A load control is necessary (but not sufficient) for the successful implementation of such a policy: for if there are too many active programs, pages will be taken from the resident set of the least recently run program (whose pages will tend to occupy the lowest stack positions), whereupon that program when run will soon experience a page fault. Even if the load is properly controlled, a running program that fortuitously generates a page fault before referencing much of its locality set will not have moved many locality pages to the top of the LRU stack whereupon, when next it is run, it may find part of its locality set missing from memory. And this state may persist, as the program is now unable to reference many pages at all between page faults. For these reasons, this type of policy has been found very susceptible to thrashing, and it is sometimes precarious to expect such

policies to perform better than fixed partition policies [B4,D6,R2]. Similar remarks apply to a policy based around a single ("global") FIFO list; it is worth noting, however, that a FIFO-based policy with load control was used successfully on the M44 system [B2,B3].

A variant of the "global LRU" policy above is based on a usage bit u and a changed bit c associated with every resident page. The bit u is set to 1 by the addressing hardware on any reference to the given page, and is cleared to 0 by the memory management routine. The bit c is set to 1 by the addressing hardware on any write-reference to the given page, and is cleared when the page is loaded or when a copy is made in the paging I/O store. At intervals, the memory management routine scans all resident set pages and maintains them in four lists according to the possible values of the bits (u,c). At a page fault, the first page of the first nonempty list in the ordering

$$(u,c) = [(0,0),(0,1),(1,0),(1,1)]$$

is selected for replacement. This policy, which approximates LRU [S3], is subject to the same problems when used for multiprogramming.

A well known example of a V2-policy is used in Multics. It combines elements of the global LRU and global FIFO policies. It is sometimes referred to as FINUFO (first in not used first out) [C5,D1]. All the resident set pages are linked in a circular list with a pointer designating the "current position", and each has a usage bit set by the hardware when the page is referenced. Whenever a page fault occurs, the memory policy advances the current-position pointer around the list, clearing set usage bits, and stopping at the first page whose usage bit

is already clear: this page is selected for replacement. A program whose locality set is resident will evidently fare well under this policy, since it will be able to continue setting all its usage bits between the times when the memory policy examines them. However, an active program whose locality set is not loaded, or which is accorded continuing low priority for use of the processor, will tend to lose pages under this policy. The success of this policy will depend on its being carefully coordinated with the scheduler, which must control the load and ensure that all active programs have an equal chance to use the processor. There is no performance data comparing this against a fixed partition or V1-policy.

Another example of a V2-policy is the "AC/RT" procedure suggested by Belady and Tsao [B4]. Associated with each active program P_i are variables, AC_i and RT_i, whose values are updated at each page fault of P_i. The "activity count" AC_i registers that fraction of its resident set referenced since P_i's last page fault; the "round trip frequency" RT_i registers that fraction of the last K page faults (K a parameter) of P_i which caused the recall of the most recently replaced page. A high value of AC_i indicates that P_i is making effective use of its resident set. A high value of RT_i indicates a high frequency of mistakes in replacement decisions. The decision rule for replacement, used on a page fault of P_i is summarized as: If RT_i is low, replace a page from the resident set of P_i; otherwise, replace a page from that $Z_j(t)$, $j \neq i$, for which AC_j is lowest. Belady and Tsao discuss how to select threshold values to tell when AC and RT are "high" and "low", and infer (but do not test) that this policy will perform better than policies in Class V1 and the global LRU or FIFO policies in Class V2. As with other V2-policies,

however, load control must augment the AC/RT procedure: too high a level of multiprogramming can force the persistent state in which all the AC_i are low and the RT_i are high -- the state of thrashing.

We have returned repeatedly to the need for V2 (and V1) policies to be augmented by a load control. Operational experience with Multics and CP-67 indicates that an effective combination of a V2-policy and load control can be designed [C5,R2]. The same was true of the V1 biasing policy on the M44 [B2,B3]. With proper load control, V2-policies will tend to be better than V1-policies because their capability of reallocating pages from resident sets that are too large for their locality sets, to resident sets that are too small, permits a higher level of multiprogramming without an increase of paging. Since heavy-demand conditions are not at all uncommon, one arrives at the conclusion to include the load control and locality estimation explicitly in the memory policy -- i.e., at Class W.

Class W Policies

As has been noted, W-policies have two distinguishing features. First, the resident sets are precisely the estimates of the current locality sets of active programs. Moreover, the locality estimate of P_i is formed by observing the behavior of P_i only -- it is not influenced by the activity of any other program P_j. (Contrast this with the V2-policies, in which a resident set $Z_i(t)$ is a function not only of the activity of P_i itself, but of the activities of other programs as

well.) Second, load control is inherent in the definition of W-policies, since program activation and deactivation decisions must be consistent with the requirement that locality set estimates of all active programs must be resident.

The definition of W-policies implies the existence of a memory reserve of size R(t) -- i.e., a set of pages not in any resident set (see eq. (4.4)). To improve memory utilization, some systems allocate the reserve R(t) to an n+1st program P_0, whereupon $Z_0(t)$ is a subset of P_0's locality set and page faults by any P_i ($0 \leq i \leq n$) cause pages to be preempted from $Z_0(t)$. In case $z_0(t)=R(t)=0$, P_0 is considered to be automatically deactivated, and the lowest priority program among $P_1,\ldots,P_n$ assumes the role of P_0. System thrashing cannot occur in this case: although P_0 is the only program without a full locality set present, its page faults are not permitted to preempt pages from other resident sets and, accordingly, the feedback among paging rates necessary for thrashing does not exist (see W2 and S5).

The most extensively studied example of a Class W policy uses the moving window working set $W_1(t,T)$, defined previously, as the locality estimator. Numerous experiments have shown the ease with which one can find a suitable value for the window size T so that the working set is indeed a reliable estimator of a program's locality [C2,F2,H1,R1, S5,S7]. However, the estimator is not perfect [P3].

Morris reports how the MANIAC II computer implements a close approximation of the moving window working set, by associating hardware timers with each page of main memory and arranging to run a given page's timer only when the program owning that page is running on the processor; all at modest cost [M2]. A method of approximating a working set by

examining usage bits at the ends of time slices appeared successful in preliminary tests of the RCA Spectra 70/46 [W1]. A similar procedure was used on the Grenoble CP-67, for which extensive test data show enormour improvements in performance over the V2-policy used on the standard CP-67 [R2]. Another similar procedure has been used successfully on at least one TSS system [D7]. A method using two window sizes to define three states of a page (in, partly in, and out of the working set) has been reported successful in UNIVAC's VMOS [F2]. These and other practical and successful implementations show definitively that W-policies are neither difficult nor expensive to implement; they are at worse marginally more expensive than V2-policies and give significantly better performance -- if only because they are able to operate at a maximal level of multiprogramming without thrashing.

An interesting variant to the fixed-window-size working set defined above has been studied by Chu and Opderbeck using extensive simulations [C2,O1]. Their procedure, known as PFF (page fault frequency), recomputes a program's working set at each page fault time t of that program, using the time interval since the prior page fault of that program (time t') as a window. The computation requires merely the examination of usage bits. Unfortunately, should the current window t-t' fortuitously be small, few usage bits will have been set; since this will cause the next page fault interval to be short, the state of the working set underestimating locality will persist. Protection against this is easily achieved. If the interval t-t' is smaller than a given threshold T_0, the incoming page is added to the resident set but no replacement is made (though the usage bits are cleared). The acronym PFF arises

since $1/T_0$ has the interpretation of the maximum allowable mean rate (frequency) of page faults. The resident set defined by PFF for program P_i at a page fault time t, to be in effect until P_i's next page fault, is

$$Z_i(t) = \begin{cases} W_i(t,t-t'), & t-t' \geq T_0 \\ Z_i(t')+r(t), & \text{otherwise} \end{cases} \tag{4.8}$$

where t' is the time of the prior page fault and r(t) is the (missing) page referenced at time t. Besides the usage bits, the full implementation evidently requires only a timer register in the processor to compute t-t'. Chu and Opderbeck's studies indicate that T_0 can easily be chosen so that PFF is indistinguishable from a fixed-window working set [C2], and that PFF used as a W-policy is significantly better than certain LRU-type policies from Classes V1 and V2 [O1].

Figure 8 suggests why a W-policy will be better than a fixed partition policy, as long as programs are run in a region of the fault rate curve in which WS is superior (cf. Fig. 5). Let z_i denote the resident set size for program P_i under a fixed partition using LRU separately for each resident set. As long as $z_i > x_{0i}$, there will exist a point $w_i < z_i$ corresponding to a mean working set size under which the program would achieve the same fault rate as under the LRU policy. Setting $W = w_1+\ldots+w_n$, this means that the average level of multiprogramming could be increased approximately by the ratio M/W without increasing the system fault rate over the original fixed partition policy, which in turn implies an increase in processing efficiency (cf. eq. (2.11)).

Figure 9 suggests the application of the above principle to conclude that a W-policy will be superior to a V1-policy. The three points on the vertical line through $\bar{x}_i$ depict the relation (4.7) given earlier for V1-policies. As long as $X_i > x_{0i}$, there will exist a point

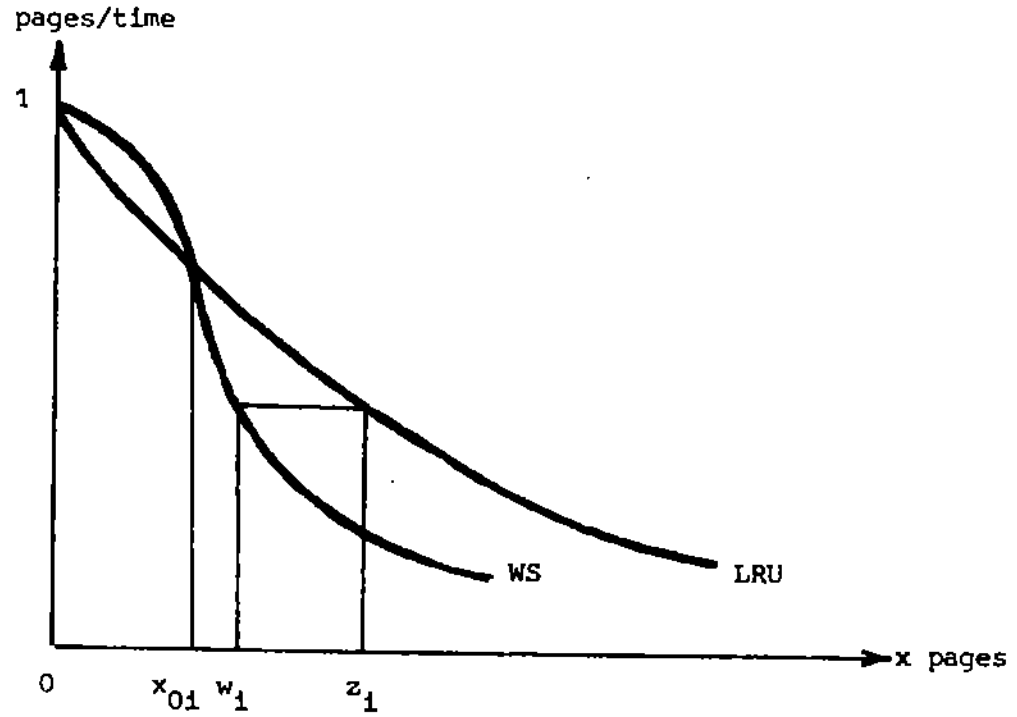

Figure 8. Comparison of W-policy and fixed partition.

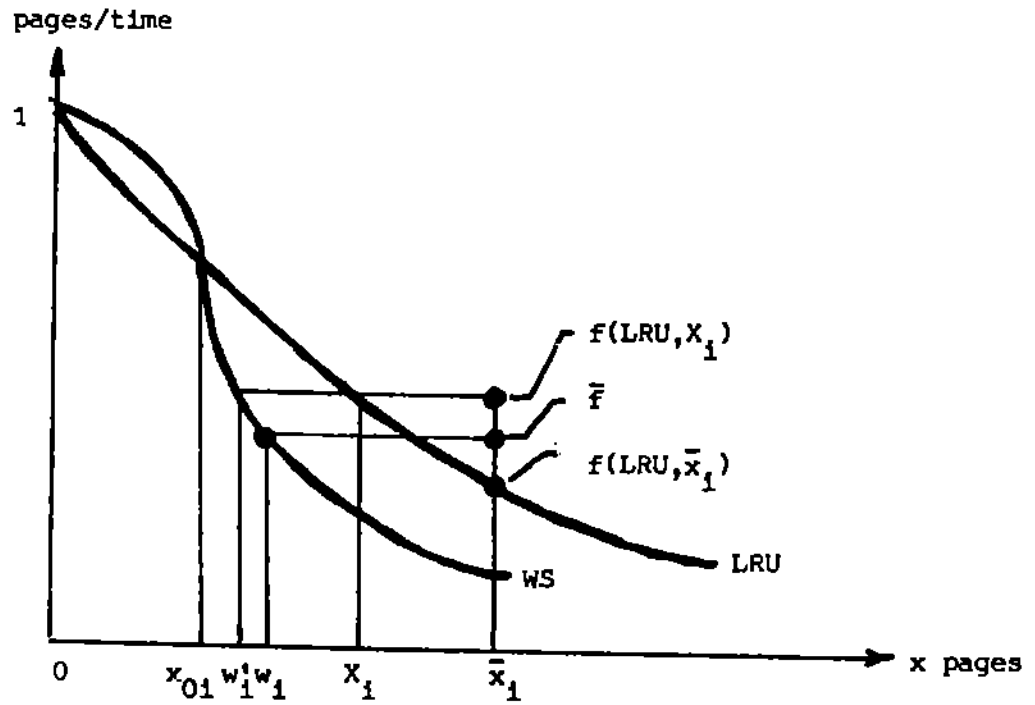

Figure 9. Comparison of W and V1 policies.

$w_i < X_i$ at which $f(WS,w_i) = \bar{f}_1$. Setting $W = w_1+\ldots+w_n$, the average level of multiprogramming could be increased by approximately the ratio M/W and yield, as before, higher efficiency. (The W-policy produces less an improvement over the V1-policy than over the fixed partition. Define w_i' so that $f(WS,w_i') = f(LRU,X_i)$, note that $w_i' < w_i$, and that $W'=w_1'+\ldots+w_n'$ < W. Therefore the ratio M/W' is larger than M/W.)

The discussion above shows that working set policies increase processing efficiency over other policies. However, they have been shown to improve other measures as well. Chu and Opderbeck, for example, show that the "space-time cost" (integral of resident set size over time) satisfies

(4.9) $$ST(WS,T) < ST(LRU,x)$$

for all x, and all T in a very wide range [C2]. In fact, the minimum difference between the two sides of this inequality ranged 10-30%, the greater differences being directly correlated to large coefficient of variation (ratio of standard deviation to mean) in locality set size. The function ST(LRU,x) had a sharp minimum, while ST(WS,T) had a very wide and flat minimal region; therefore, for x injudiciously chosen, the space time cost difference may far exceed the 30% figure just quoted. Coffman and Ryan studied two measures of storage utilization, overflow probability and mean amount by which demand exceeds resident set size, comparing a working set partition against a fixed equipartition [C3]. With respect to these measures, the working set partition was always at least slightly better, and significant differences would exist for larger coefficient of variation in locality size.

The W-policies appear superior by many measures, their superiority is associated with changing locality size in programs, and the degree of superiority increases as the coefficient of variation in locality size increases.

5. CONCLUSIONS

The first part of this paper explained a network representation of a typical multiprogrammed computer system, and used it to establish properties used later in the paper: a) Increasing the load (i.e., level of multiprogramming) without changing system or program parameters increases processing efficiency. b) Decreasing the paging rate for fixed load increases processing efficiency. c) Paging rates will generally increase with increasing load because of the fixed total memory constraint. This implies an optimum load, above which efficiency drops rapidly (thrashing). Load control is necessary. d) Processing efficiency is a suitable measure of system performance, since throughput is directly proportional to it and response time inversely proportional to it.

The second part of the paper explained basic properties and measures of program behavior. The principal observations are: a) The fault rate function of LRU is frequently observed to be convex, while the lifetime function frequently has a convex/concave shape. b) The fault rate function of WS (working set) is frequently observed to be significantly below that of LRU, a direct proof of locality size variation during program execution.

The third part of the paper explained a classification of multiprogrammed memory management policies, then used the results of the previous sections, together with information from the literature, to establish a ranking among five classes of policies, from worst to best:

1. Fixed partition, balanced;
2. Fixed partition, imbalanced;
3. Variable partition, no correlation with program behavior (V1);
4. Variable partition, some correlation with program behavior (V2); and
5. Variable partition, direct estimation of locality (W).

(This ranking should be interpreted to mean that, given a policy at rank i, there exists a better one at rank i+1.) The principal conclusions are: a) Imbalanced partitions are better than balanced partitions, partly because they recognize inherently different memory requirements of programs, and also because of the convex property of the lifetime function. In many cases an equipartition is the worst possible, even among programs with identical memory demand characteristics. b) Even though they do not correlate memory reallocations with program behavior, V1-policies may nonetheless improve over fixed partition policies. Two factors operate: the avoidance of the equipartition, and the effect of increasing average processor demand over the virtual time of programs; both factors are attributable to the convexity of the lifetime function. c) V2-policies do better than V1-policies because they obtain better space utilization by reallocating from resident sets that are larger than contained localities to resident sets that are smaller, and because, with proper load control, they tend to keep each program's locality set present. d) W-policies do better than V2-policies because they estimate locality directly,

the estimates are independent of load and other programs' demands for memory, and they have inherent load control. Numerous studies show W-policies best according to a variety of measures. Their implementation cost is not significantly more than for V1 or V2, and the gain in performance amply rewards the investment in them.

Working set (W) policies establish a limit on the load n(t) at each time t. To the extent that these policies succeed in estimating locality sets, n(t) will approximate the optimal load $n_0(t)$. Experience shows that these policies keep the probability of thrashing (i.e., the probability that $n(t) > kn_0(t)$ for some small constant $k \geq 1$) acceptably small. In contrast, V1 and V2 policies have no direct method of estimating a proper load level. Typically they establish a prior limit N on the load (sometimes with adjustments in N inversely with the system paging rate); since N must be chosen so that the thrashing probability (the probability that $kn_0(t) < N$) is low, the system runs much of the time at suboptimal efficiency. In other words, the more precise load control of working set policies is of itself a significant reason for their success.

All the arguments, and all the experiments, used to demonstrate the superiority of working set policies rely directly on, or are correlated directly with, significant variations in program locality size over virtual time. Though there has been considerable work on modeling program behavior (e.g., C4, G2, S5, S7), none of it has so far produced a working model in which locality set size variation is accounted for. Many experiments show that many programs exhibit a marked propensity for two or more particular working set sizes [G2,H1,R1], and that working set fault

rates are signficantly less than LRU fault rates over a wide range of memory constraints; these observations cannot be accounted for under the assumption of fixed locality size. The next iteration in the process of program behavior modeling must be the development of techniques for representing locality set size variation.

The viability of working set policies and locality-based program models appears assured.

APPENDIX 1: COMPUTATION OF FAULT RATE FUNCTIONS

Outlined here are computationally efficient methods for finding LRU and WS fault rate functions, for a given reference string $\underset{\sim}{R} = r(1)\ldots r(k)\ldots r(K)$. The techniques are treated fully in C4, D4, M1, and S4.

The LRU Algorithm. The LRU stack at virtual time k is a vector $\underset{\sim}{s}(k) = (s_1,\ldots,s_{q(k)})$ of distinct pages, in which $1 \leq i < j \leq q(k)$ implies that page s_i was more recently referenced than s_j, and q(k) is the number of distinct pages referenced through time k. The initial stack $\underset{\sim}{s}(0)$ is empty. The stack distance d(k) of the reference r(k) is i if r(k) is at the i^{th} position in stack $\underset{\sim}{s}(k-1)$, and is ∞ if r(k) is not in stack $\underset{\sim}{s}(k-1)$. If r(k)=y, the new stack $\underset{\sim}{s}(k)$ is related to the former by

$$\underset{\sim}{s}(k) = \begin{cases} (y,s_1,\ldots,s_{i-1},s_{i+1},\ldots,s_{q(k-1)}), & \text{if } d(k) = i \leq q(k-1) \\ (y,s_1,\ldots,s_{q(k-1)}), & \text{if } d(k) = \infty \end{cases}$$

Note in the second case $q(k)=q(k-1)+1$.

Since the LRU algorithm always replaces the least recently used page, it follows that the pages resident in an x-page memory managed by LRU at time k are precisely the first x entries in the stack $\underset{\sim}{s}(k)$; and moreover that a page fault occurs at time k if and only if d(k) > x. Therefore, the fault rate function f(LRU,x) is the fractional number of distances that exceed x. To calculate f(LRU,x), one must process the references r(1)r(2)..., computing the stacks $\underset{\sim}{s}(0)\underset{\sim}{s}(1)\underset{\sim}{s}(2)\ldots$, and recording the occurrences of stack distances d(1)d(2)... in the counters c[1:N] and c[∞]. (The number of program pages is N.) When this is done, c[i]

counts the number of virtual times k at which d(k)=i. Once the stack distance counts have been determined, the number of page faults for an x-page page memory is c[x+1]+...+c[N]+c[∞]; therefore, the LRU fault rate is computed from the recursion formula

$$f(LRU,N) = \frac{c[\infty]}{K}$$

$$f(LRU,x-1) = \frac{c[x]}{K} + f(LRU,x), \quad x = N,N-1,\ldots,1$$

To obtain the counts, the following procedure is used:

```
c[1:N,∞] := 0; stack[1:N] := Ø;                    [initialize]
for k:=1 to K do
   y := r(k);                                      [next reference]
   i := 1; candidate := stack[1];
   while candidate ≠ y and candidate ≠ Ø do         [search and update
      exchange(candidate, stack[i+1]);              stack for entry y]
      i := i+1;
   end
   if candidate = Ø                                [update proper
      then c[∞] := c[∞]+1                           counter]
      else c[i]  := c[i] +1;
   stack[1] := y;                                  [put referenced
                                                    page atop stack]
end
```

The WS Algorithm. Associate with $\underset{\sim}{R} = r(1)\ldots r(k)\ldots r(K)$ a sequence of backward distances $\underset{\sim}{B} = b(1)\ldots b(k)\ldots b(K)$, in which $b(k)=i$ implies $r(k-i)=r(k)$ and $r(k')\neq r(k)$ for $k-i < k' < k$; take $b(k) = \infty$ if $r(k)$ is the first reference to a page. In other words, $b(k)$ is the interval since the prior reference to page $r(k)$. (For example, if $\underset{\sim}{R}$ = 1 2 3 2 3 1, $\underset{\sim}{B} = \infty\ \infty\ \infty\ 2\ 2\ 5$.) The next reference $r(k+1)$ is missing from the working set $W(k,T)$ if and only if $b(k+1) > T$. Define the counters $c[1:K]$ and $c[\infty]$ to record the occurrences of backward distances; thus $c[i]$ counts the number of distinct virtual times k at which $b(k)=i$. Analogous to LRU, the missing page rate for pure working set memory allocation is defined by the recursion formula

$$m(K) = \frac{c[\infty]}{K}$$

$$m(T-1) = \frac{c[T]}{K} + m(T), \quad T = K,K-1,\ldots,1$$

To obtain the counts, this algorithm can be used:

```
c[1:K,∞] := 0; time[1:N] := 0;
for k:=1 to K do
   y := r(k);
   if time[y]=0
      then c[∞] := c[∞]+1
      else i := k - time[y]
           c[i] := c[i]+1;
   time[y] := k;
end
```

The working set size at time k is denoted by w(k,T) and the mean working set size by

$$s(T) = \frac{1}{K} \sum_{k=1}^{K} w(k,T).$$

Define $\Delta(k,T)$ to be 1 if r(k) is missing from W(k-1,T) and 0 otherwise. Then note $w(k,T+1) = w(k-1,T) + \Delta(k,T)$; substituting into the definition of s(T),

$$s(T+1) = \frac{1}{K} \sum_{k=1}^{K} w(k-1,T) + \frac{1}{K} \sum_{k=1}^{K} \Delta(k,T)$$

$$= \frac{1}{K} \sum_{k=1}^{K} w(k,T) - \frac{w(K,T)}{K} + \frac{1}{K} \sum_{k=1}^{K} \Delta(k,T).$$

Recognizing the last term as a definition of the missing page rate m(T), we find the recursion formula for calculating mean working set size:

$$s(0) = 0$$

$$s(T+1) = s(T) + m(T) - \frac{w(K,T)}{K}, \qquad T = 0,1,\ldots,K-1.$$

Finally, the fault rate function is denoted by f(WS,x) and is given parametrically by

$$f(WS, s(T)) = m(T), \quad T = 0,1,\ldots,K$$

Examples. Consider the three reference strings over a 10-page program,

$$\underset{\sim}{R}_1 = 01\ldots9(9\ldots1001\ldots9)^{10}$$

$$\underset{\sim}{R}_2 = 01(01)^{20}23\ldots9(23\ldots9)^{20}$$

$$\underset{\sim}{R}_3 = 01(1001)^{10}23\ldots9(9\ldots3223\ldots9)^{10}$$

All have length K=210. $\underset{\sim}{R}_1$ represents a program using a single 10-page locality; since $\underset{\sim}{R}_1$ has the property that, at any time the page with the largest stack distance is also the one with the maximum forward distance, LRU is optimal for $\underset{\sim}{R}_1$ [M1]. In contrast, $\underset{\sim}{R}_2$ and $\underset{\sim}{R}_3$ represent programs which have two disjoint localities $\{0,1\}$ and $\{2,3,\ldots,9\}$. In $\underset{\sim}{R}_3$, LRU is optimal just as in $\underset{\sim}{R}_1$. Figure 10 shows the fault rate curves for LRU and WS for these strings.

It is observed that LRU is always better for $\underset{\sim}{R}_1$, WS is at least as good for $\underset{\sim}{R}_2$, and WS is better for $\underset{\sim}{R}_3$ provided $x > 6.6$. The superiority of WS over LRU for (certain ranges of x in) $\underset{\sim}{R}_2$ and $\underset{\sim}{R}_3$ directly results from these two strings' exhibiting two distinct phases over different size localities: For suitable choices of the window size T, the working set measures the locality set exactly (as long as the window is contained within a phase) so that the only paging occurs during locality transitions. However, the average working set size is less than that of the larger locality; LRU operating at that same memory size produces page faults continuously in that phase because that locality will not fit into the available space. It is especially important to note that, because of its ability to adapt its space requirement to varying program locality, WS is capable of improving over an optimal fixed space algorithm (such as LRU applied to $\underset{\sim}{R}_3$). Similar observations have been made in practice [P1,P2,P3].

Figure 11 shows the lifetime functions for the three strings. Each exhibits the characteristic convex/concave shape. The concave region for the LRU lifetime function begins at the maximum locality size. The concave region for the WS lifetime function begins at approximately the

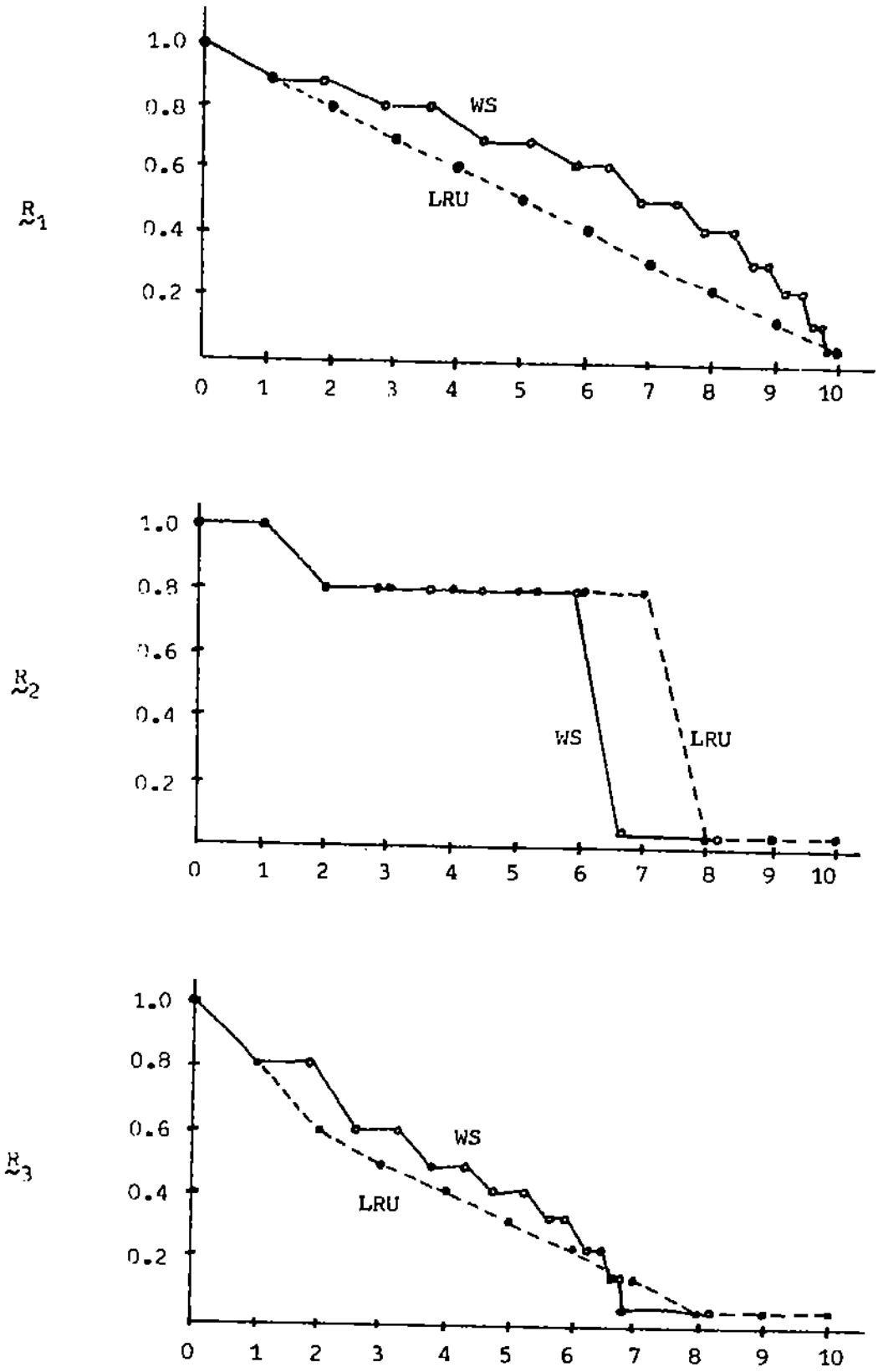

Figure 10. Examples of fault rate functions.

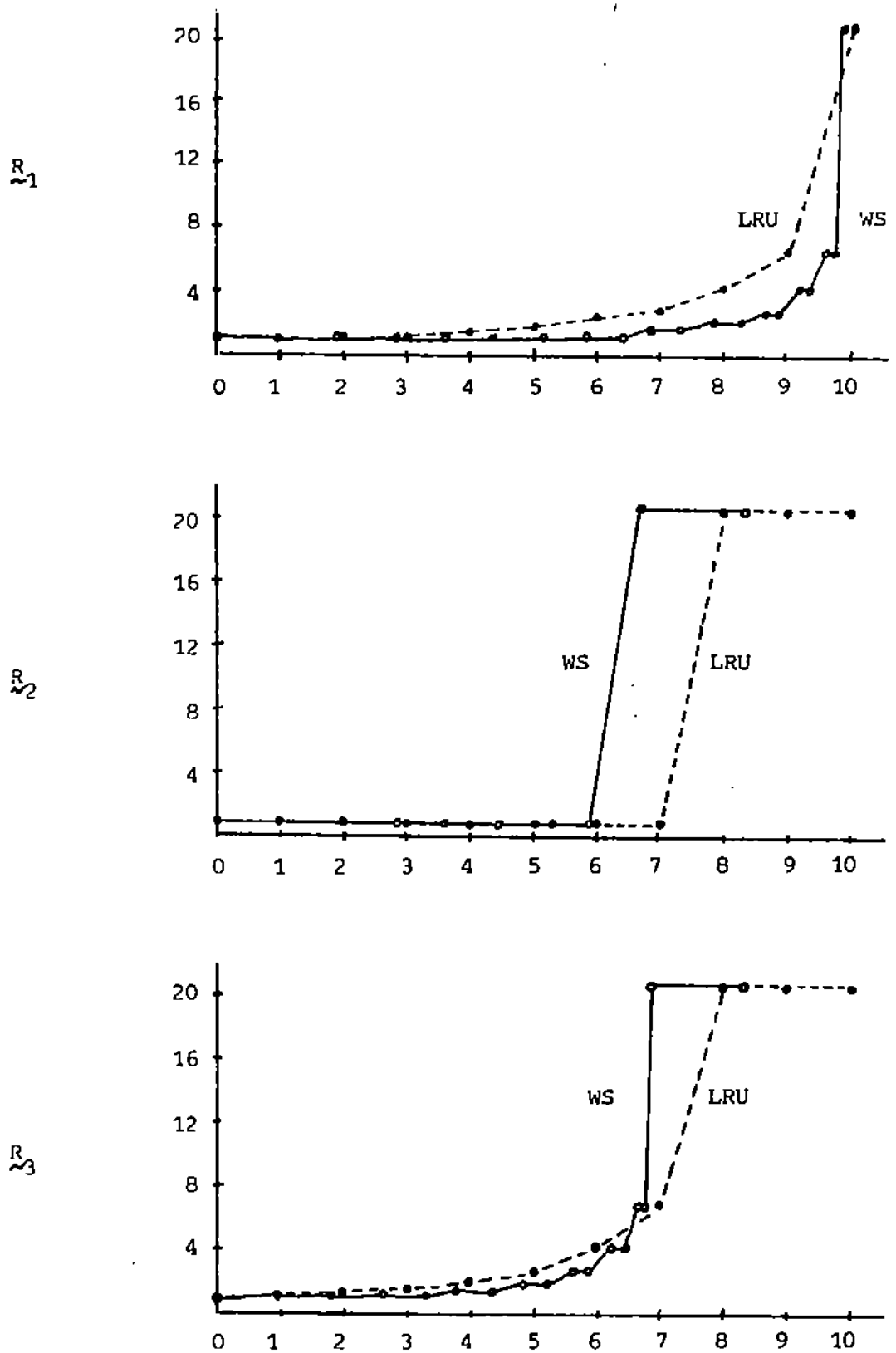

Figure 11. Examples of lifetime functions.

virtual time average locality size (for strings R_2 and R_3, the average locality size is computed as$(2\cdot42 + 8\cdot168)/210 = 6.8$).

APPENDIX 2: ANALYSIS OF V1 POLICIES [See also D5]

General Properties. Let $\{x_k\}$ be a sequence containing at least two distinct values such that the function h(x) is convex for $\min\{x_k\} \leq x \leq \max\{x_k\}$, and let $\{a_k\}$ be a set of positive weights that sum to 1. A well known property of convex functions is

$$(1) \qquad \sum_k a_k h(x_k) > h\left(\sum_k a_k x_k\right).$$

Our objective is proving relation (4.7) of the text which states that, given a partition $\underset{\sim}{X} = (X_1,\ldots,X_n)$ one may construct a V1-policy under which for each program P_i,

$$(2) \qquad f_i(X_i) > \bar{f}_i > f_i(\bar{x}_i),$$

where f_i is the fault rate function, $\bar{f}_i$ is the mean virtual time fault rate under the V1-policy, and $\bar{x}_i$ is the mean virtual time memory allocation under the V1-policy. The lefthand inequality requires the convexity of the lifetime function, the righthand one the convexity of the fault rate function.

Let $t_0=0$ and $t_1 t_2 \ldots t_r$ denote a sequence of successive page fault time on a system's processor. Let r_i denote the number of faults generated by program P_i in the observation interval $(0,t_r]$, and note that

$$(3) \qquad r = \sum_{i=1}^{n} r_i .$$

Let x_{ik} denote the memory allocation of program P_i in the processing interval just preceding its kth page fault ($1 \leq k \leq r_i$). Each x_{ik} is assumed to lie in the convex region of L_i. The mean virtual time interval from the k-1st to the kth page fault in P_i is taken to be the lifetime $L_i(x_{ik})$. (This is, in fact, an approximation. As will be discussed shortly, however, it does not affect the conclusions.) Under these assumptions, the mean lifetime in P_i over the observation interval is

$$\bar{L}_i = \frac{1}{r_i} \sum_{k=1}^{r_i} L_i(x_{ik}), \tag{4}$$

and the mean fault rate over this interval is $\bar{f}_i = 1/\bar{L}_i$. Define the quantity

$$X_i = \frac{1}{r_i} \sum_{k=1}^{r_i} x_{ik} \tag{5}$$

which is the mean memory allocation measured at page fault times. We shall show shortly how the scheduler can choose the allocations x_{ik} so that X_i is the same as the resident set size of P_i according to the given fixed partition $\underset{\sim}{X}$.

Under the given fixed partition $\underset{\sim}{X}$, the mean lifetime interval of program P_i is $L_i(X_i)$, so that the mean system lifetime interval is the total processing time consumed divided by the total number of page faults:

$$\bar{L} = \sum_{i=1}^{n} \frac{r_i}{r} L_i(X_i). \tag{6}$$

Under a V1 policy satisfying (5), the mean system lifetime interval is

$$\bar{L}' = \sum_{i=1}^{n} \frac{r_i}{r} \bar{L}_i , \tag{7}$$

since $r_i \bar{L}_i$ is (from (4)) the total time consumed by P_i. Applying (1) to (4),

(8) $$\bar{L}_i > L_i(x_i).$$

Since lifetime is the reciprocal of fault rate, this establishes the lefthand inequality of (2). Applying (8) to (6) and (7), we have $\bar{L}' > \bar{L}$, which implies that the relative utilization of the paging I/O station satisfies

(9) $$R_p' = S/\bar{L}' < S/\bar{L} = R_p ,$$

where S is the mean service time at the paging I/O station; together with eq. (2.14) of the text, this implies that a V1-policy satisfying (5) must increase processor utilization over the fixed partition $\underset{\sim}{x}$.

As noted in the text after eq. (2.9), the use of $\bar{L}'$ and $\bar{L}$ in (9) is an approximation. The processor utilization is in reality a function of all the lifetime intervals, not just their mean. Ghanem [G1] and Spirn [S6] have shown that, when L_i are sufficiently convex, $\bar{L}' > \bar{L}$ will imply the increase in utilization as argued here. Spirn showed that observed lifetime functions do usually have the required convexity; hence, our simple argument is sufficient to justify our conclusions.

To establish the righthand inequality of (2), define $T_i = r_i \bar{L}_i$ as the mean lifetime interval in P_i, and note that

(10) $$\bar{x}_i = \sum_{k=1}^{r_i} \frac{L_i(x_{ik})}{T_i} x_{ik} .$$

Using the definition of fault rate as reciprocal of lifetime, and assuming the fault rate function is convex, we have the inequalities as desired:

$$(11)\quad \bar{f}_i = \frac{r_i}{T_i} = \sum_{k=1}^{r_i} \frac{L_i(x_{ik})}{T_i} f_i(x_{ik}) > f_i\left(\sum_{k=1}^{r_i} \frac{L_i(x_{ik})}{T_i} x_{ik}\right) = f_i(\bar{x}_i).$$

Since f_i is decreasing, relation (11) implies $\bar{x}_i > X_i$. However $\bar{x}_i > X_i$ can be shown directly, even if f_i is not convex: observe that there exists u such that $L_i(x) \geq L_i(u)$ if and only if $x \geq u$ and consider

$$(12)\quad \bar{x}_i - X_i = \sum_{k=1}^{r_i} x_{ik}\left(\frac{L_i(x_{ik})}{T_i} - \frac{1}{r_i}\right) > u \sum_{k=1}^{r_i}\left(\frac{L_i(x_{ik})}{T_i} - \frac{1}{r_i}\right) = 0.$$

It was noted prior to eq. (4) that the use of $L_i(x_{ik})$ is an approximation. The reason is that the virtual time interval between the k-1st and kth page faults may be interrupted by $p \geq 0$ file I/O requests, so that P_i in fact experiences during this interval a resident set size sequence $y_0 y_1 \cdots y_p$, in which $y_0 \geq y_1 \geq \cdots \geq y_p$ and $x_{ik} = y_p$. However, this implies that $L_i(x_{ik})$ underestimates the true lifetime in this interval; therefore $\bar{L}_i$ underestimates the true mean lifetime, and relations (8) and (9) remain valid. Moreover, $\bar{x}_i$ underestimates the true virtual time resident set size, and relation (12) remains valid. Finally (11) remains valid: for we can interpret $\bar{f}_i$ as the true value, observe that the second equality in (11) is an identity, then recall that $\bar{x}_i$ is an underestimate and f_i is decreasing. The errors introduced by this approximation are not likely to be large, especially in systems with $R_f \leq 1$: for the mean file I/O service time is usually 10 times the mean paging I/O service time, and $R_f \leq 1$ implies that p=0 at least 90% of the time.

Implementation. Eqs. (6) and (7) allow for the possibility of an arbitrary scheduling discipline over the observation interval -- the ratios r_i/r reflect the relative priorities given to the programs. For FIFO scheduling, each of these ratios will tend to be 1/n.

A V1-policy satisfying (5) for a given partition $\underset{\sim}{X}$ may be approximated arbitrarily closely using an adaptive procedure. Let D_i denote the relative deviation of the mean resident size of P_i from the desired X_i:

$$D_i = \frac{1}{r_i} \sum_{k=1}^{r_i} \frac{x_{ik}-X_i}{X_i} . \tag{13}$$

The estimator D_i can be updated on each page fault of P_i by the statements

$$\begin{aligned} D_i &:= (D_i r_i + (z_i-X_i)/X_i)/(r_i+1), \\ r_i &:= r_i+1, \end{aligned} \tag{14}$$

where z_i is the resident set size at the page fault. The memory allocation decision rule can be implemented as a two-phase repeating procedure. During the "converge phase" a page fault in P_i will result in a page being removed from P_j, where j=i if $D_i \geq 0$, and j is the

index of the program with largest positive deviation if $D_i < 0$; the effect of a memory reallocation during this phase will be to reduce the total relative deviation of the memory partition from the desired $\underset{\sim}{X}$. During the "diverge" phase a page fault in P_i will result in a page being removed from P_j, where j is the index of the program with smallest absolute deviation; the effect of a main memory reallocation in this phase will be an increase in the total relative deviation of the memory partition from the desired $\underset{\sim}{X}$. At the end of a pair of diverge/converge phases a partition sequence that conforms to (5) will have been generated, whereupon the V1-policy has generated higher processor utilization than the fixed partition $\underset{\sim}{X}$.

If $\underset{\sim}{X}$ is an equipartition, any symmetric memory reallocation procedure with FIFO scheduling -- such as the cyclically permuted favored state under the "biasing" policy [B2,B3] -- is sufficient to produce a V1-policy improving over $\underset{\sim}{X}$.

Fixed Imbalanced Partitions. It is possible for a fixed imbalanced partition to improve over an equipartition. Let $\underset{\sim}{X}$ be a given partition in which at least two resident sets have different size. Suppose that FIFO scheduling is used at the processor and paging I/O stations. Under these assumptions $r_i/r = 1/n$ and $x_{ik}=X_i$ for each i. The mean lifetime for $\underset{\sim}{X}$ will be larger than that of the equipartition if

$$\sum_{i=1}^{n} L_i(X_i) > \sum_{i=1}^{n} L_i(\tfrac{M}{n}), \tag{15}$$

which is certainly true if there exists a convex function L passing through the points $L_i(X_i)$ and $L_i(\frac{M}{n})$; if fact, if $L_i=L$ for all i, every

imbalanced partition is better than the equipartition. (See also G1, S6, pertaining to networks with different queueing disciplines.)

Example. Consider two active programs with the same fault rate and lifetime functions:

x	f(x)	L(x)
10	100/S	S/100
20	4/S	S/4
30	1/S	S

where S is the mean paging I/O station service time. Suppose that the file I/O station is unused (R_f=0). Consider a (nondemand paging) variable partition policy that allocated memory according to the partition sequence (10,30)(30,10) for equal numbers of page faults in each partition; and a fixed partition (20,20). Using the formulae given earlier with S=10:

measure	partitions (10,30)(30,10)	(20,20)
$\bar{L}$	5.05	2.50
$\bar{f}$	0.20	0.40
$\bar{x}$	29.8	20.0
X	20.0	20.0
U_0	0.43	0.24

The utilization U_0 was computed according to Buzen's method [B7] and verified by simulation. A linear interpolation between f(20) and f(30) gives $f(\bar{x})$ = 0.11 and verifies relation (2).

By the symmetry of the example, the imbalanced fixed partition (10,30) will produce U_0=0.43, while the balanced partition (20,20) will produce U_0=0.24, verifying that balanced partitions may be less efficient than imbalanced ones.

See Ref. D5 for another view of this analysis.

APPENDIX I: NEAR OPTIMAL PARTITIONS

Figure 12 suggests why a working set policy is capable of generating a near-optimal partition. Consider a set of active programs having the same lifetime function L under a working set policy, and suppose that L does not increase much for x larger than the inflection point y_0. (Specifically, assume that the slopes satisfy $L'(x_0) < L'(w_0)$, for w_0 to be defined below.) For a partition $\underset{\sim}{x}$, the mean lifetime (assuming FIFO queueing in the network) is

$$L(\underset{\sim}{x}) = \frac{1}{n} \sum_{i=1}^{n} L(x_i).$$

Our objective is finding a partition that maximizes $L(\underset{\sim}{x})$.

Consider a working set partition with average resident sizes $\underset{\sim}{w} = (w_0, w_1, \ldots, w_n)$ in which $w_i = y_0$ for $1 \leq i \leq n$ and $w_0 = M-(w_1+\ldots+w_n) < y_0$ -- i.e., one in which the reserve memory w_0 is allocated to an n+1st program. Consider any other partition $\underset{\sim}{v} = (v_0, v_1, \ldots, v_n)$, such as might be generated under another variable partition policy. If any $v_i > y_0$, $\underset{\sim}{v}$ cannot maximize $L(\underset{\sim}{x})$ since decreasing v_i to y_0 and reallocating the pages $v_i - y_0$ to the program with smallest v_j will increase $L(\underset{\sim}{v})$. Assuming all $v_i < y_0$, then all $v_i > w_0$, else $v_0+\ldots+v_n=M$ is impossible. Since $L(\underset{\sim}{w})$ lies on the chord connecting the points w_0 and y_0 on the L curve, and since $L(\underset{\sim}{v})$ lies below the chord between v_0 and v_n on the L curve, it is clear that $L(\underset{\sim}{w}) > L(\underset{\sim}{v})$. In other words, $\underset{\sim}{w}$ is an optimal partition for n programs and will maximize processing efficiency.

An optimal partition for n-1 programs will have $L(\underset{\sim}{u}) \geq L(y_0) > L(\underset{\sim}{w})$; whether it produces higher processing efficiency than $\underset{\sim}{w}$ depends on whether the increase in L offsets the decrease in load. A partition $\underset{\sim}{u}'$

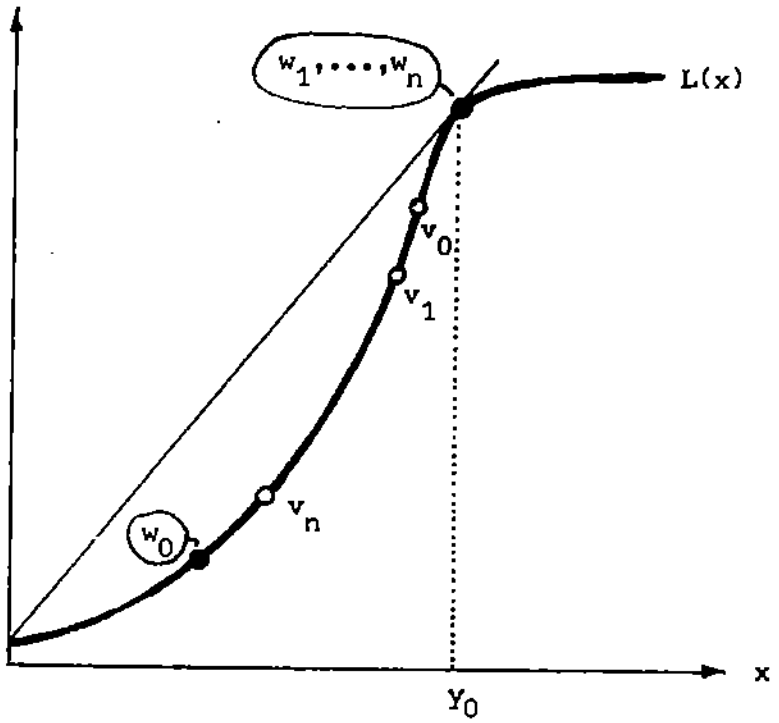

Figure 12. Approximating an optimal partition.

for fewer than n-1 programs will have $L(\underset{\sim}{u}') \simeq L(\underset{\sim}{u})$; since it has smaller load than $\underset{\sim}{u}$, it is less efficient. It is not difficult to see that a partition $\underset{\sim}{v}'$ for n+1 programs (in which $v_i < y_0$) has $L(\underset{\sim}{v}') < L(\underset{\sim}{w})$; whether the processing efficiency for $\underset{\sim}{v}'$ exceeds that of $\underset{\sim}{w}$ depends on whether the effect of increased load offests that of decreased lifetime.

The point is, the partition w will approximate an optimal partition and an optimal load. It remains only to recall that the inflection point y_0 is approximately the mean locality size of the program, which can be approximated by a working set policy for a wide range of window size. (See also G1.)

Bibliography

B1. Belady, L. A. "A Study of Replacement Algorithms for Virtual Storage Computers." IBM Sys. J. 5, 2 (1966), 78-101.

B2. Belady, L. A. "Biased Replacement Algorithms for Multiprogramming." IBM T. J. Watson Research Center Note NC 697 (March 1967).

B3. Belady, L. A. and C. J. Kuehner. "Dynamic Space Sharing in Computer Systems." Comm. ACM 12, 5 (May 1969), 282-288.

B4. Belady, L. A. and R. F. Tsao. "Memory Allocation and Program Behavior under Multiprogramming." Proc. Computer Science & Statistics: 7th Annual Symposium on the Interface, Iowa State University (October 1973), 72-78.

B5. Brandwajn, A. "A Model of a Time Sharing Virtual Memory System Solved using Equivalence and Decomposition Methods." Acta Informatica 4 (1974), 11-47.

B6. Buzen, J. P. "Computational Algorithms for Closed Queueing Networks with Exponential Servers." Comm. ACM 16, 9 (September 1973), 527-531.

C1. Chamberlin, D. D., S. H. Fuller, and L. Y. Liu. "A Page Allocation Strategy for Multiprogramming Systems with Virtual Memory." IBM T. J. Watson Research Center Report RC 3848 (May 1972).

C2. Chu, W. W. and H. Opderbeck. "The Page Fault Frequency Replacement Algorithm." Proc. AFIPS Conf. (1972 FJCC), 597-609.

C3. Coffman, E. G. Jr., and T. J. Ryan, Jr. "A Study of Storage Partitioning using a Mathematical Model of Locality." Comm. ACM 15, 3 (March 1972), 185-190.

C4. Coffman, E. G. Jr. and P. J. Denning. Operating Systems Theory. Prentice-Hall (1973), Ch. 6-7.

C5. Corbato, F. J. "A Paging Experiment with the Multics System." In Ingard, In Honor of P. M. Morse, M.I.T. Press (1969), 217-228.

D1. Denning, P. J. "Resource Allocation in Multiprocess Computer Systems." M.I.T. Project MAC Report MAC-TR-50 (May 1968).

D2. Denning, P. J. "The Working Set Model for Program Behavior." Comm. ACM 11, 5 (May 1968), 323-333.

D3. Denning, P. J. "Virtual Memory." Computing Surveys 2, 3 (September 1970), 153-189.

D4. Denning, P. J. and S. C. Schwartz. "Properties of the Working Set Model." Comm. ACM 15, 3 (March 1972), 191-198.

D5. Denning, P. J. and J. R. Spirn. "Dynamic Storage Partitioning." Proc. 4th ACM Symposium on Operating Systems Principles (October 1973), 74-79.

D6. Denning, P. J. "Comments on a Linear Paging Model." Proc. ACM SIGMETRICS Symposium on System Performance Evaluation (October 1974).

D7. Doherty, W. "Scheduling TSS/360 for Responsiveness." Proc. AFIPS Conf. (1970 FJCC), 97-111.

F1. Ferrari, D. "Improving Locality by Critical Working Sets." Comm. ACM 17, 11 (November 1974), 614-620.

F2. Fogel, M. H. "The VMOS Paging Algorithm." ACM SIGOPS Operating System Review 8, 1 (January 1974), 8-17.

G1. Ghanem, M. Z. "The Lifetime Function Shape and the Optimal Memory Allocation." IBM T. J. Watson Research Center Report (September 1973).

G2. Ghanem, M. Z. and H. Kobayashi. "A Parametric Representation of Program Behavior in a Virtual Memory." IBM T. J. Watson Research Center Report (September 1973).

H1. Hatfield, D. J. and J. Gerald. "Program Restructuring for Virtual Memory." IBM Sys. J. 10, 3 (1971), 168-192.

M1. Mattson, R. L., D. Slutz, I. Traiger, J. Gecsei. "Evaluation Techniques for Storage Hierarchies." IBM Sys. J. 9, 2 (1970), 78-101.

M2. Morris, J. B. "Demand Paging Through Utilization of Working Sets on the MANIAC II." Comm. ACM 15, 10 (October 1972), 867-872.

M3. Muntz, R. R. "Analytic Modeling of Interactive Systems." Proc. IEEE Interactive Computer Systems(June 1975).

O1. Opderbeck, H. and W. W. Chu. "Performance of the Page Fault Frequency Algorithm in a Multiprogramming Environment." Proc. IFIP Congress 1974.

P1. Prieve, B. G. "Page Partition Replacement Algorithm." Ph.D. Thesis, Dep't Elec. Engrg. & Computer Science, University of California, Berkeley (December 1973).

P2. Prieve, B. G. and R. S. Fabry. "Evaluation of a Page Partition Replacement Algorithm." Technical Report, Bell Laboratories, Naperville, Illinois (October 1973).

P3. Prieve, B. G. and R. S. Fabry. "An Optimal Variable Space Page Replacement Algorithm." Technical Report, Bell Laboratories, Naperville, Illinois (May 1974).

R1. Rodriguez-Rosell, J. "Experimental Data on How Program Behavior Affects the Choice of Scheduler Parameters." Proc. 3rd ACM Symposium on Operating System Principles (October 1971), 156-163.

R2. Rodriguez-Rosell, J. and J.-P. Dupuy. "The Design, Implementation and Evaluation of a Working Set Dispatcher." Comm. ACM 16, 4 (April 1973), 247-253.

R3. Ryan, T. A. Jr. and E. G. Coffman Jr. "A Problem in Multiprogrammed Storage Allocation." IEEE Trans. Comp. C-23, 11 (November 1974), 1116-1122.

S1. Saltzer, J. H. "A Simple Linear Model of Demand Paging Performance." Comm. ACM 17, 4 (April 1974), 181-185.

S2. Sayre, D. "Is Automatic 'Folding' of Programs Efficient Enough to Displace Manual?" Comm. ACM 12, 12 (December 1969), 656-660.

S3. Shaw, A. The Logical Design of Operating Systems. Prentice-Hall (1974).

S4. Slutz, D. R. and I. Traiger. "A Note on the Calculation of Average Working Set Size." Comm. ACM 17, 10 (October 1974), 563-565.

S5. Spirn, J. R. "Program Locality and Dynamic Memory Management." Ph.D. Thesis, Dep't Elec. Engrg., Princeton Univ. (March 1973).

S6. Spirn, J. R. "A Model for Dynamic Memory Allocation in a Paging Machine." Proc. 8th Princeton Conf., Dep't Elec. Engrg., Princeton Univ. (March 1974).

S7. Spirn, J. R., P. J. Denning, and J. E. Savage. "Models for Locality in Program Behavior." Acta Informatica (1975), to appear.

W1. Weizer, N. and G. Oppenheimer. "Virtual Memory Management in a Paging Environment." Proc. AFIPS Conf. (1969 SJCC), 234ff.

W2. Wilkes, M. V. "The Dynamics of Paging." Computer J. 16, 1 (February 1973), 4-9.